(2)

ETL-0520

AD-A212 806

Spatial Data Structures for Robotic Vehicle Route Planning

Michael J. Black
David L. Milgram
Sharon O. Cioffi
Patrice Gelband

Advanced Decision Systems
1500 Plymouth Street
Mountain View, California 94043-1230

December 1988

Approved for public release; distribution is unlimited.

Prepared for:

U.S. Army Corps of Engineers
Engineer Topographic Laboratories
Fort Belvoir, Virginia 22060-5546

E T L

U.S. ARMY ENGINEER TOPOGRAPHIC LABS
ETL

Destroy this report when no longer needed.
Do not return it to the originator.

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

The citation in this report of trade names of commercially available products does not constitute official endorsement or approval of the use of such products.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) ADS TR-3185-01	5. MONITORING ORGANIZATION REPORT NUMBER(S) ETL-0520
6a. NAME OF PERFORMING ORGANIZATION Advanced Decision Systems 6b. OFFICE SYMBOL (If applicable)	7a NAME OF MONITORING ORGANIZATION U.S. Army Engineer Topographic Laboratories
6c. ADDRESS (City, State, and ZIP Code) 1500 Plymouth Street Mountain View, California 94043-1230	7b. ADDRESS (City, State, and ZIP Code) Fort Belvoir, Virginia 22060-5546
8a. NAME OF FUNDING / SPONSORING ORGANIZATION Humphreys Engineer Center Support Activity 8b. OFFICE SYMBOL (If applicable) CEHEC	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER DACA72-87-C-0015
8c. ADDRESS (City, State, and ZIP Code) Kingman Building (Bldg. 2593) Telegraph & Leaf Roads Fort Belvoir, Virginia 22060-5580	10 SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)

Spatial Data Structures for Robotic Vehicle Route Planning

12. PERSONAL AUTHOR(S)

Michael J. Black, David L. Milgram, Sharon O. Cioffi, & Patrice Gelband

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Final Technical	FROM 9/87 TO 10/88	1988 December	116

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Route Planning, Data Structures, Robotic Vehicles, Terrain Representations

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This is the final report for the Phase II SBIR contract, "Spatial Data Structures for Robotic Vehicle Route Planning." The report describes the work completed during Phase II and discusses the directions for future research.

The goal of the Phase II SBIR contract was to investigate techniques and tradeoffs for representing digital terrain information in a computer environment. The long-term goal of this research is to build a Spatial Data Structure Development System (SDSDS) to serve as the infrastructure base for terrain analysis applications.

The Phase II contract addressed the following issues: 1) implementation of common terrain representations, 2) implementation of common spatial operations, 3) design of a methodology for evaluating the performance of spatial operations, 4) evaluation of the implemented representations and operations, and 5) initial design of testbed on which the SDSDS would be built.

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL Richard Rosenthal	22b TELEPHONE (Include Area Code) (202) 355-3653 22c. OFFICE SYMBOL CEETL-RI-T

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. Introduction

1.1 Background

This is the final report for the Phase II SBIR Contract No. DACA72-87-C-0015, "Spatial Data Structures for Robotic Vehicle Route Planning" (ADS Project Number 3185). It describes the work completed during Phase II, and discusses the directions for future research.

The goal of this project is to investigate techniques and tradeoffs for representing digital terrain information in a computer environment. That goal will be realized in an intelligent Spatial Data Structure Development System (SDSDS) intended for use by Terrain Analysis applications programmers with little or no knowledge of specific spatial representations.

The representation of spatial information is a key element in any solution to a terrain analysis challenge such as route planning. In as much as the same spatial information can be represented in many different data structures, the choice of representation will often determine both the ease with which a solution can be found and implemented and the (time and space) efficiency of that solution.

Terrain analysts, however, are not experts in the computational complexity of spatial data structures. Therefore, the problem is to design and build an infrastructure to support terrain analysis applications which "hides" the details of representations and manipulations of those representations from the analyst.

A system capable of serving the analyst this way will perform many varieties of operations on the spatial data. For example, a robotic vehicle route planning application should focus attention on a number of specific issues:

- processing incomplete or multiple resolution information
- providing incremental access (scrolling) and updating of spatial information as the route develops
- time-space tradeoffs

The work performed by this contract addresses the issue of estimating and evaluating the complexity of representations and algorithms.

1.2 Objectives

The long-term goal of this research is to build a Spatial Data Structure Development System (SDSDS) to serve as the infrastructure base for Terrain Analysis applications. Towards this end, Phase II has addressed the following issues:

- implementation of common terrain representations,
- implementation of common spatial operations,
- design of a methodology for evaluating the performance of spatial operations,

- evaluation of the implemented representations and operations,
- initial design of testbed on which the SDSDS would be built.

1.3 Overview of the Technical Approach

Our approach is to view the spatial processing portion of a terrain analysis application as a separable layer of manipulations and computations over spatial representations. For example, the semantic rule "Company HQ must be within .5 miles of a major road" can be interpreted as having (at least) two distinct levels of context - the application level represents what is known about application objects - HQ's and major roads; the spatial data level is concerned with the representation of point-like objects, linear networks and the analysis of their spatial relationships.

Another aspect of our approach, which is hinted at above, is to design and implement object-oriented representations. Since objects can point to other objects, the above example would have an instance of a road object pointing to an instance of a linear structure object (its geometric description). The linear structure object might have several representations (e.g. k-d tree, edge quad-tree) each at multiple resolutions. The value of this extensive structuring is to hide from the user the details of representation and to permit the infrastructure itself to decide which representations will be most efficient or effective for a given process request.

The third aspect of our approach is to select efficient methods and representations for an operation based on both analytic and empirical models of the algorithmic complexity. Since these models are available as part of the infrastructure, the SDSDS system will be able to intelligently predict performance of algorithmic sequences and thereby optimize the application (within the accuracy of the prediction models). The development of a methodology for generating these models has been a major focus of the Phase II effort.

1.4 Focus of Study and Structure of the Final Report

The technical approach taken in Phase II is outlined in Chapter 2. This chapter states the problem, defines key concepts, and describes the research tasks undertaken. Chapter 3 presents a description of the spatial representations and operations chosen for study. The algorithms for each spatial operation are described in detail. The main technical results of the Phase II research are presented in Chapter 4. This chapter contains descriptions of the studies in algorithm complexity, data complexity, and performance modeling. The analysis of this chapter threads together the various studies and draws conclusions based on their results. Chapter 5, the final technical chapter, presents an initial design of a future spatial reasoning testbed based on the results of Chapter 3 and Chapter 4. The conclusions of Chapter 6 summarize the Phase II effort and make recommendations for future research.

2. Technical Approach

2.1 Statement of the Problem

There are numerous ways of representing terrain data in current computer systems, such as arrays, records, and files. Each of these represents a particular point in the tradeoff between execution time and memory space. Automatic route planning poses particularly strong demands on any such representation because it requires large amounts of space to contain high resolution maps of large areas of the earth as well as requiring fast response times for interactive or autonomous use. There is currently no satisfactory data structure that meets those demands.

The tradeoff between time and space is reflected in the levels of memory storage in modern computer systems. Generally the faster the access to the unit of memory, the more limited its storage capabilities. Traditionally, this range has gone from registers through main memory through secondary (disk) memory to off-line (tape) memory. For this application, the space requirements are such that main memory by itself is insufficient. Similarly, the time demands remove the possibility of using off-line storage. Thus some form of secondary storage will be necessary. There are several ways that secondary storage can be utilized to augment main memory such as virtual memory, files, or databases. These again involve tradeoffs between time and space and also interact with the particular form and use of the data.

An efficient, on-line data structure that supports both effective encoding of data (to meet the space requirements) and decoding of data (for the time requirements) will have a significant payoff. Interactive (man-in-the-loop) systems require high bandwidth data flow due to the large amount of graphics necessary to communicate with the user. Even more stringent are the real-time requirements of autonomous robotics systems. Both applications would benefit from success in this area.

Terrain data has many uses besides automatic route planning. These include navigation, perceptual prediction, aerial photograph and satellite imagery analysis, land management, and cartography. Many of these have similar time and space requirements and so could benefit from an effective method of data representation. In addition, there are numerous applications that need to manipulate large amounts of data quickly (i.e., are in similar points in the time/space domain). These include computer graphics, signal understanding, modeling, seismic interpretation, and numerous civilian applications.

The long range goals of this project are to develop:

1. an object-oriented representation for the storage of terrain data
   - supported by efficient spatial data structures and algorithms
   - supporting data provided by sources with different resolutions
   - capable of representing objects at multiple resolutions
   - and which can be incrementally updated efficiently

2. a hardware and software testbed supporting the object-oriented representation with tools to manipulate the data

3. an infrastructure layer on the testbed which permits a researcher or developer to rapidly prototype and evaluate systems which reason about terrain data

4. a methodology which supports the automated optimization of spatial operations.

The primary emphasis is on automatic route planning, but the research draws on and considers any applicable knowledge of terrain analysis and corresponding research issues determined by and identified for other applications.

The object-oriented representation permits the division of raw terrain data (elevations plus thematic overlays) into a network of schemata which are represented as objects. The schemata will be constructed so as to represent entities that are semantically significant (e.g., spatial structures for roads, fields, rivers) and thus permit efficient access to relevant data. While the technique is suitable for many terrain analysis applications, emphasis will be on uses of terrain suited for route planning.

Schemata are made up of three parts: the geometry of the terrain feature location, non-geometric properties, and relations to other schemata. The route planning domain has the advantage that it is not strictly three-dimensional. That is, no point on the earth's surface can have more than one elevation. Thus, an augmented two-dimensional ($2\frac{1}{2}D$) representation is sufficient (where one considers the data from a downward-looking orthographic viewpoint) to capture the topology of the scene. Object geometry can then be represented naturally as points, two-dimensional curves, or regions along with the elevations at selected feature points. There are numerous ways of encoding point, curve, and region information; we choose a representative set for consideration.

Geometry is further characterized by two related properties: faithfulness and resolution. Faithfulness refers to the ability to decode the raw data from the encoded form. The resolution of the data indicates the level of detail at which it is represented. These properties have a strong effect on the space/time tradeoff since high resolution, highly faithful representations require considerable space (and time to process). Multi-resolution schemes are important in the route planning application so that the space/time tradeoff can be explicitly controlled. At each resolution, one would like the ability to manipulate faithfulness so that it is adequate to the task but not wasteful of storage.

The properties of schemata are either facts supplied with the raw data (e.g., surface material) or information derived during the encoding phase (e.g., maximum curvature). Thus an example schema for a road might include its length, width, average elevation, and a straight-line approximation to the center line of the road (at several resolutions) which was augmented with the elevations at those points.

The final component of the schemata is their relationships to other schemata. These are essentially indexing shortcuts which tradeoff additional space for more efficient processing capability. For example, if the relationship "adjacent-to" is precomputed, pointers representing adjacency can be stored in the schemata. Thus,

we trade off additional storage for reduction of runtime computation to determine adjacency. In the route planning application useful relations include adjacency, containment, and overlap. The resolution hierarchy is also a type of relation. In the example above, a particular road can be "adjacent" to several other schemata. It might also "contain" intersections.

The schemata, as linked by their relationships, form a semantic network. Such networks have a long history in artificial intelligence and many graph search algorithms have been developed that can effectively utilize their structure.

Clearly, the choice of schemata should be tied to the particular application. An object-oriented representation has the strong advantage that the connection can be made at an explicit, semantic level. This will lead to a more flexible, manageable system, useful across a range of different applications.

2.2 Key Concepts

The following are concepts central to the Phase II study. These concepts we become building blocks with which we will define a testbed environment which supports a terrain application developer. One of the main tasks of such a testbed is to take high level descriptions of terrain operations and execute them in the most efficient way given the available algorithms and data structures. This optimization will require the ability to model and predict the performance of algorithms. The following concepts provide the foundation for this modeling.

2.2.1 Algorithm Complexity

Algorithm complexity refers to the *analytic complexity* of an algorithm [Bass 78]. This is the standard algorithm analysis which describes how an algorithm will behave as its input changes. (usually increasing size). This is often expressed in *order-form*, e.g. O(N log N) means that the algorithmic time complexity for large values of the inps Length N, varies as N log N.

2.2.2 Representation Complexity

Representation complexity is tied to the notion of algorithm complexity. The feature or features of a representation on which the algorithm complexity is based determine the representation complexity of the input to the algorithm. So, for example, if an algorithm's complexity is tied to the number of edges in its polygonal input, then the number of edges in a given polygon object determines its representation complexity.

Notice that an object with multiple representations can be viewed as having different representation complexities depending on the input assumptions of the algorithm using it as input.

2.2.3 Inherent Data Complexity

Whereas representation complexity describes the complexity of some data in a given representation, it would be simple to have a measure of data complexity which is independent of representation. The *inherent data complexity* of an object is dependent on properties of that object which are invariant under transformations between representations. A goal of this research is to define such a complexity measure and test its usefulness as a complexity measure. To be a good measure of complexity it must:

- be a good predictor of algorithm performance,
- be computable from any representation,
- be independent of representation,
- be quickly computed.

Algorithm performance is usually specified in terms of representation complexity, hence if the inherent complexity measure is closely correlated with various representation complexities it will likely be a good predictor of algorithm performance.

2.2.4 Performance Model

The knowledge of the representation complexity of the input to an algorithm as well as the algorithm's analytic complexity, is sufficient to predict the relative costs of running a given algorithm on a set of inputs. What cannot be gleaned from this knowledge is the relative speeds of different algorithms on different representations of the same input datasets. We cannot in practice, for example, predict whether polygon intersection or RLE intersection will be faster for a given dataset.

To be able to compare different algorithms we must have a *performance model* of that algorithm. This model is based on the analytic complexity but takes into account the overhead involved in running a particular algorithm. For example if the analytic complexity of an algorithm is $O(n^2)$, the performance model might be a function like: $0.01n^2 + 3.23n + 47$. From this performance model the expected run time of an algorithm can be predicted, once the representation complexity of the input is known.

A performance model based on representation complexity is only useful if the representation complexity of the input is known. A task of the SDSDS will be to predict the performance of many algorithms on different representations of the same data, and choose an optimal approach. To do this using representation-based performance models would mean that the object would have to be converted to every representation considered. This would be prohibitively expensive.

The preferred solution, if possible, is to have an inherent data complexity measure for the data, upon which performance models are based. Then the inherent complexity of the input allows the relative expected performances of each algorithm applied to that input to be computed efficiently.

2.3 Structure of the Solution

The goal of this research is to be able to predict the performance of algorithms on a given data set without using the traditional algorithm and representation complexity. The reason that this is desirable is that in order to compute representation complexity, the dataset must be instantiated in that representation. We want to be able to evaluate the cost of possible processing paths without actually doing any of the processing.

For the inherent data complexity measure to be useful, the computation of the measure must be more efficient to compute than to instantiate all possible representations and compute their complexities. If there are many representations to choose from it is likely that a simple image measure will be less expensive to compute.

The above goals suggest a research approach:

- Select Spatial Representations
  - analyze representation complexity
- Select Spatial Operations
  - analyze algorithmic complexity
- Define an Inherent Data Complexity Measure
- Select Test Data
  - choose real data
  - chose data of varying complexity
  - compute inherent data complexity of data
  - compute representation complexity of data
- Evaluate Complexity Measure
  - compare inherent data complexity to human expectations
  - compare inherent data complexity with representation complexity
- Evaluate Algorithms
  - generate performance model based on representation complexity
  - generate performance model based on inherent data complexity
  - compare performance models

This analysis will yield the answers to the following questions:

- Have we defined a good measure of inherent complexity?
- How are data and representation complexity related?

- Is data complexity a good predictor of algorithm performance?

If the answer to the above questions is "yes" then we have a methodology for analyzing algorithms and predicting performance at run-time. This ability will provide a foundation for the SDSDS testbed.

3. Data Structure and Algorithm Descriptions

3.1 Selection Criteria for Data Structures and Algorithms

A major objective of the Phase II effort of this project has been to study the relative efficiency of a set of methods when implemented using different representations. The approach to this objective is to determine the theoretical efficiency of each method/representation pair, based upon assumptions about the spatial data, and then to measure actual performance based upon real data, comparing the actual and theoretical results. The sets of chosen methods allow comparison of the use of different representations for a single method. The result of such a study is to determine which representation is best for each step of a process. This determination may show that the best design for the entire process is to use a single representation across all steps, accepting some inefficiency in some steps to gain efficiency in other steps; or, alternatively, to make a representation change between some steps; or even to carry along multiple representations of the data throughout the processing so that each step can use the representation most suited to itself. This approach supports the design of an object oriented system, since any object can include more than one representation of the data.

Out of a vast array of possible methods and representations a subset the methods was chosen to cover a wide mix of methods; the subset of representations consists of those representations most commonly used for regions. We have restricted our attention initial study was limited to regions because region operations tend to be more complex than point and curvilinear feature operations; furthermore, many point and curvilinear feature operations can be derived as special cases of region operations.

For this Phase II effort, the following methods were chosen:

1. Intersection
2. Union
3. Negation
4. Enveloping (region growing)

The following representations for regions were chosen:

1. Grid Based (Binary)
2. Run Length Encoding
3. Polygons with straight line edges

Each method/representation pair is studied by:

1. Performing a theoretical efficiency analysis of the resulting algorithm
2. Implementing the algorithm in a SUN-III/Commonlisp environment

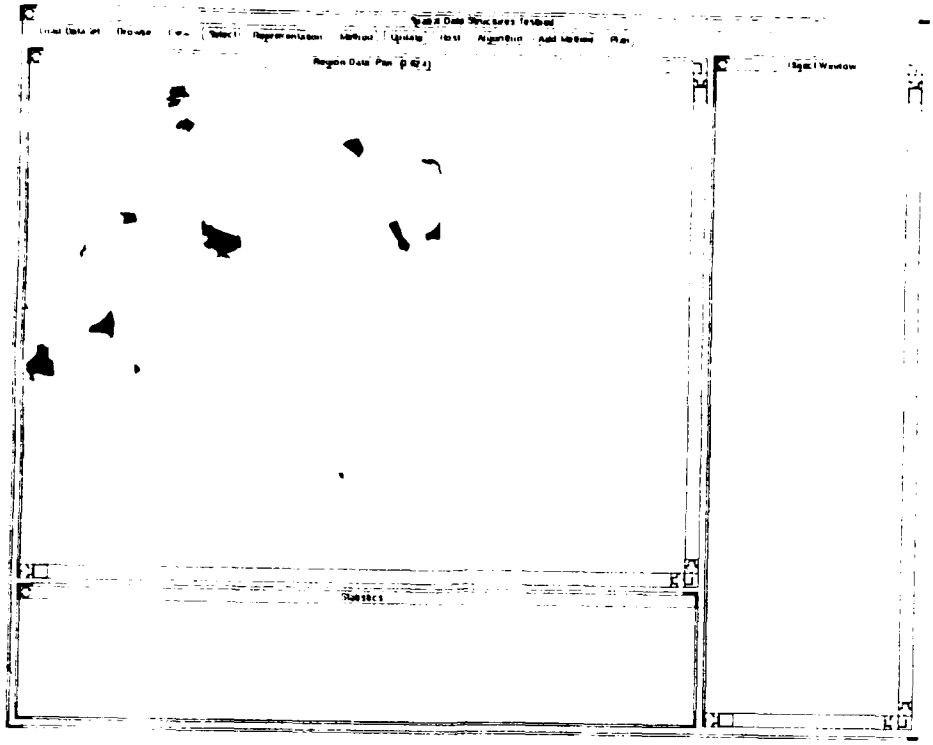


Figure 3-1: Dataset 3I

3. Measuring the performance of the algorithm on data sets of differing complexity

To illustrate the logical operations we show two region datasets (Figure 3-1 and Figure 3-2). The *intersection* of these datasets is shown in Figure 3-3. The *union* is shown in Figure 3-4. Figure 3-5 shows the negation of dataset 7A. Enveloping is best illustrated by an example of polygon enveloping. Figure 3-6 shows a portion of a dataset which has been enveloped. The inner contours correspond to the original representation and the outer contours correspond to the envelope. For this study, we treat enveloping as region-growing. That is, the envelope of a region contains the new boundary of the region as well as the region itself. Another use of enveloping operations is to construct a region corresponding some distance in either direction from the original region boundary. This type of enveloping can be computed directly for polygons or by using a combination of region growing and intersection operations for other representations.

3.2 Grid Encoded Image Operations

The most common representation of region data is *grid* format. This representation maps regions onto a discrete grid of cells. The value of a grid cell determines how it is to be interpreted. In the simplest case (and the one we implement here) a binary grid is used. Each grid cell can take on the value 1 or 0. By convention, grid cells labeled 1 indicates that the region being represented covers that cell; 0 means it does not.

A slightly more complex grid representation is a *connected-components image*. In this representation each disjoint connected region in the grid is assigned a unique

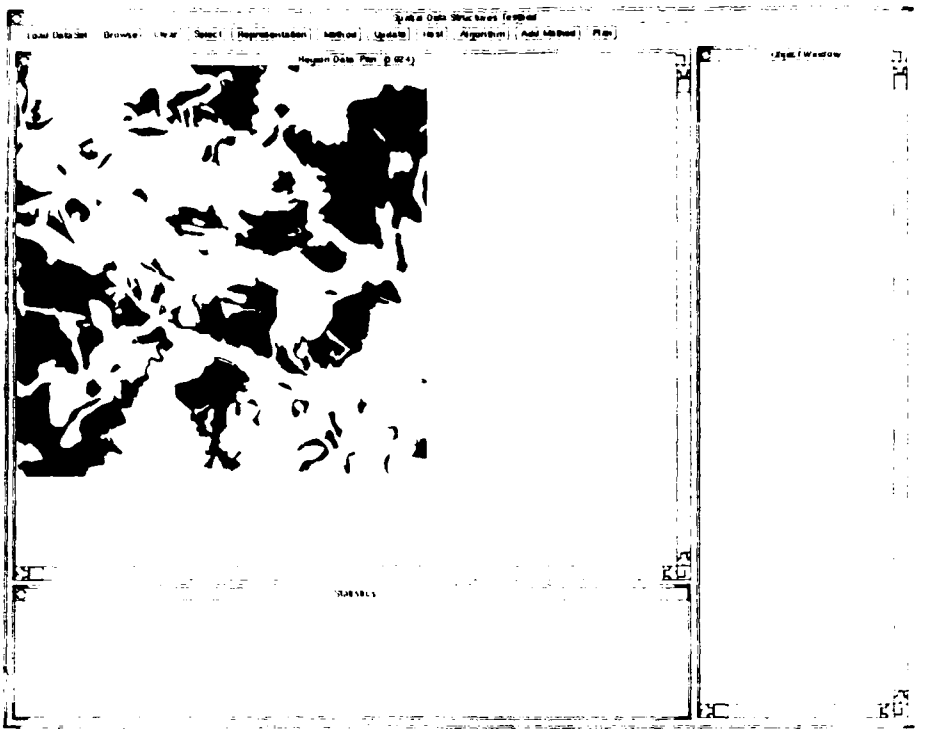


Figure 3-2: Dataset 7A

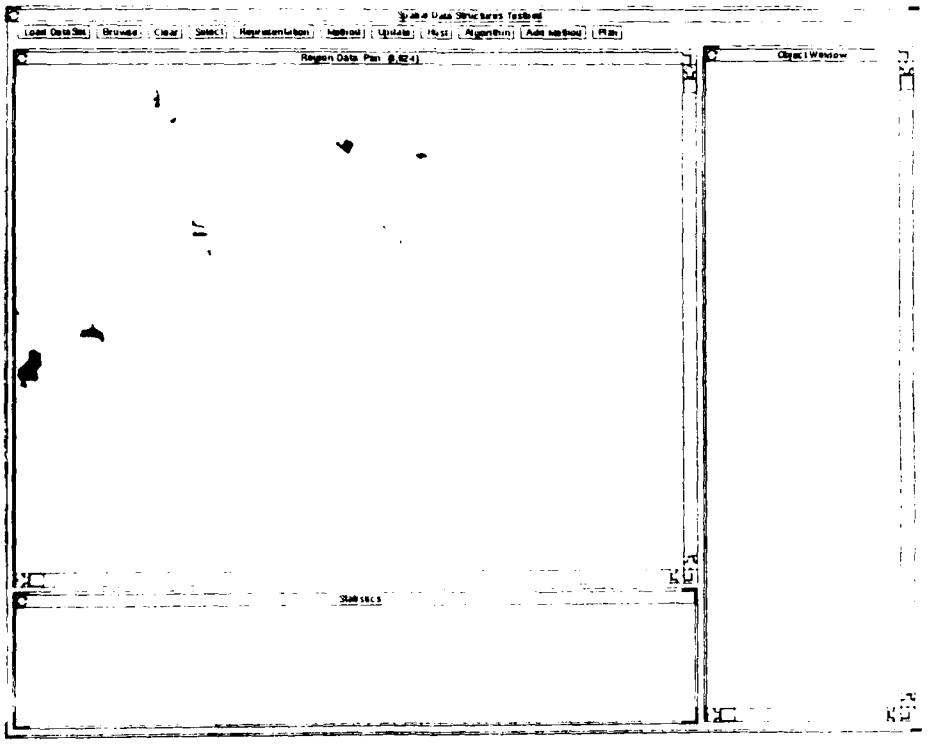


Figure 3-3: Intersection of 7A and 3I

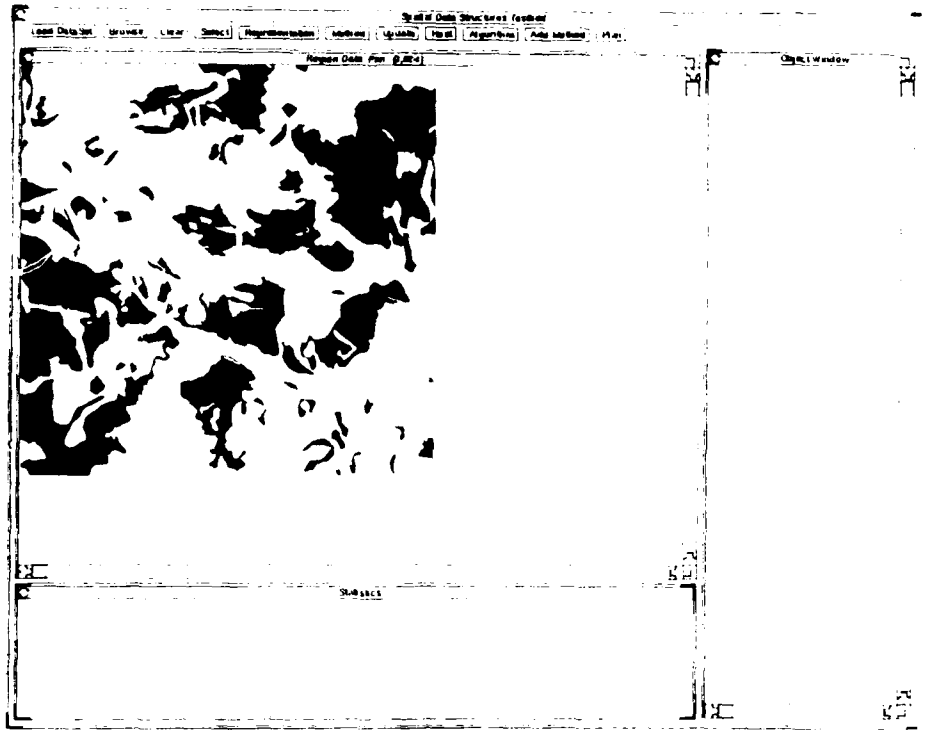


Figure 3-4: Union of 7A and 3I

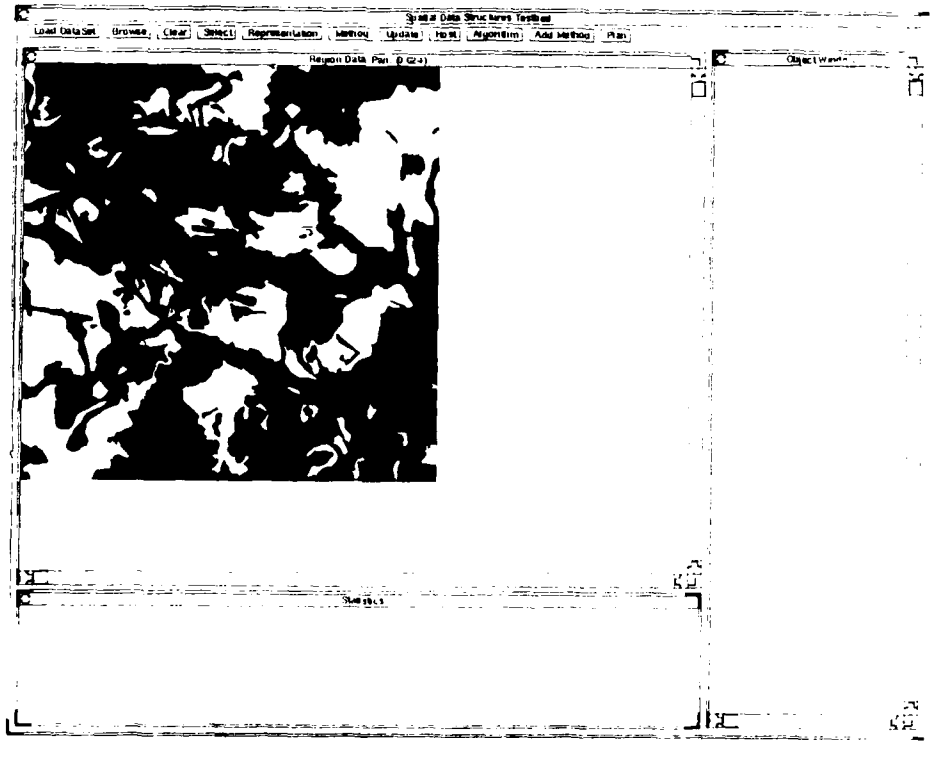


Figure 3-5: Negation (Inverse) of Dataset 7A

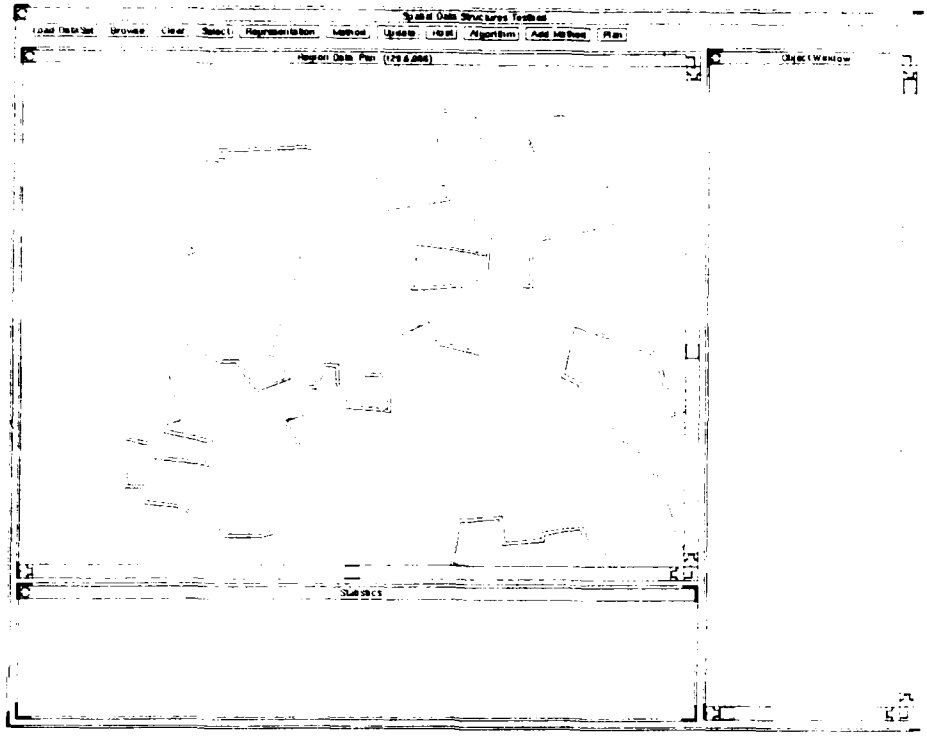


Figure 3-6: Detail of Polygon Envelope

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0																
1					1	1										
2			1	1	1	1										
3			1	1	1	1										
4			1	1	1	1				4					4	
5			1	1	1					4	4			4	4	
6			1	1						4	4	4		4	4	
7			1							4	4	4	4	4	4	
8											4	4	4	4	4	
9												4	4	4	4	
10				5	5	5	5	5								
11			5	5	5			5								
12			5	5				5								
13			5	5	5	5	5									
14					5	5										
15																

Figure 3-7: Connected-Components Grid Representation

label (as opposed to simply a 1) (see Figure 3-7). This representation encodes more information than the binary grid because disjoint connected regions can be easily determined.

While being very simple, grid representations have drawbacks as region representations. First, they are not very space efficient. Even in the case of a binary grid, a full grid is used even if only a few cells are non-zero. We will see in future sections how run-length encoding and polygonal representations exploit properties of regions to achieve a more efficient use of space. Secondly the representation does not explicitly represent objects as spatial entities. This makes the performance of certain spatial operations more difficult and time-consuming.

Many terrain databases use a grid representation, and hence the ability to perform spatial operations on grids is important. In fact grid representations are so common throughout Computer Science, many computers have special-purpose hardware for performing grid-based operations. In developing and testing grid-based algorithms we assume that no special-purpose hardware is available; this provides a fair basis of comparison with the other representations. However, in an eventual system, the distribution of algorithms for specific representations across special architectures will permit automated selection of the most efficient implementation.

3.2.1 Union and Intersection

Union and intersection operations for grids are easily defined. Assuming that the grids have the same dimensions, the corresponding pixels in each grid are examined. Intersection corresponds to a logical-AND of the corresponding pixels and union corresponds to a logical-OR. That is, a new grid is created and its grid values are the result of either an AND or an OR operation on the corresponding grid cells of the input grids.

3.2.2 Inverse

The inverse of a grid is also straightforward to define. A new grid is created with grid cells whose values are the logical-NOT of the corresponding input cells. That is, if the input grid has a 1 in cell (x, y) then the inverse grid has a 0 in cell (x, y).

3.2.3 Envelope

The envelope of grid encoded regions is calculated by constructing a square mask of width and height $2R + 1$ where R is the enveloping radius. This mask is moved over the grid and when it is centered on a non-zero cell all cells covered by the mask are set to 1. This is also called, a "grow" operator. The inverse operator, turning I's to O's is called "shark"

3.3 Run Length Encoded Images

A run length encoded image (RLE) is a simple representation of an arbitrary two dimensional shape. Run length encoded images are best used to represent shapes

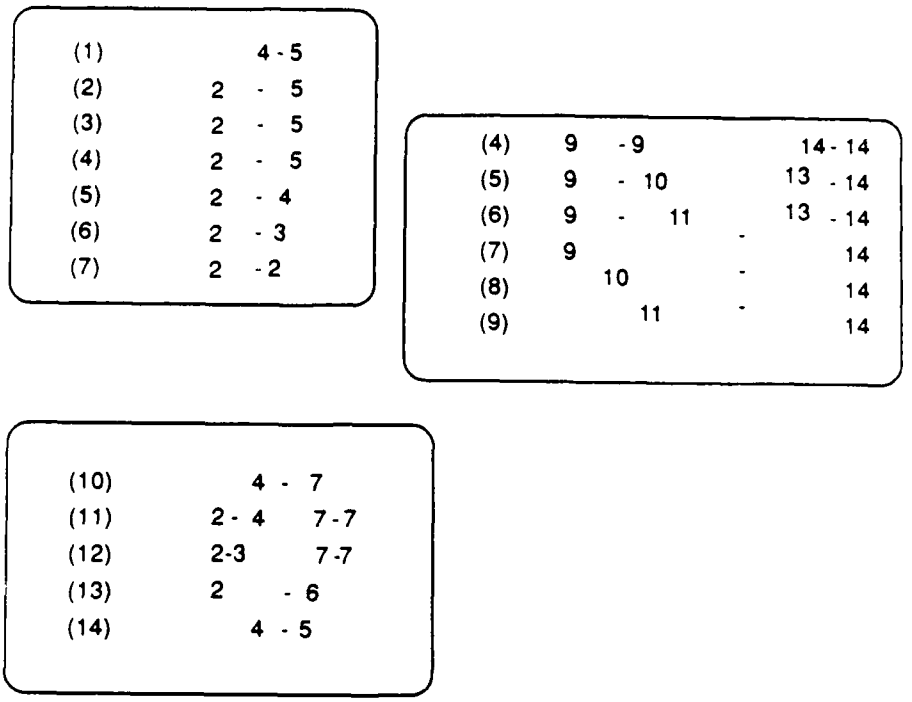


Figure 3-8: RLE Representation

that are defined with respect to a grid and are somewhat homogeneous. Typically the underlying grid corresponds to the coordinate system for an image.

Run length encoded images, *RLE*s, are represented in terms of intervals of connected feature pixels. The intervals are aligned with the row axis of the underlying grid (Figure 3-8). Each interval is called a *run*. The set of runs that all fall on one row is called a *run-list*. The complete RLE is a list of run-lists. Run-lists are sorted by increasing row index, runs are sorted by increasing column index. By definition: runs on the same row can not overlap.

Logically, the internal representation of a run length encoded image looks like this:

(run-list1 run-list2 run-list3 ... run-list*n*)

The actual RLE structure includes pointers to both ends of the each run list and pointers to the first and last run-lists. A run-list is represented like this:

row on ((start . stop) (start . stop) ...)

Runs (i.e., a cons (start . stop)) are sorted in increasing order, i.e., left to right and run-lists (i.e., (row (start . stop) ...)) are sorted from the bottom to the top of the image.

3.3.1 Union and Intersection

The algorithms for run length encoded image union and intersection are similar in structure; one would expect that most of the other set operations (e.g., Exclusive Or, Difference) would also have the following structure:

- Test for and handle special cases: zero or one RLE argument, none of the RLE arguments overlap.
- Find the union or intersection one row at a time: find all of the run-lists on each row that is overlapped by any of the RLEs and then find the union or intersection of those run-lists.
- Test and handle special cases: only one run-list on a row, none of the run-lists on a row overlap.
- Use a simple left to right sweep algorithm to find the union or intersection of all of the runs on a single row.

The remaining sections focus on the last step, i.e., finding the union or intersection of a set of run-lists when there is more than one overlapping run-list in the set.

3.3.1.1 Union (rle-or rle1 rle2 ... rle*n*)

This algorithm scans through all of the input run-lists simultaneously moving from left to right. The algorithm is driven by a loop that finds the start column and then the stop column for each run in the output run-list. The loop terminates when all of the input run-lists are empty.

(**run-list-OR** *rls*)

Find the union of the sequence of run-lists *rls*.

1. If all of the run-lists are empty then finish. If only one non empty run-list remains then append a copy of it to the output run-list and finish.
2. Initialize start and stop to the leftmost first run in *rls*.
3. Remove runs that are contained within start,stop
4. If the 1st run in any remaining run-list overlaps start,stop then update stop, remove the run, and return to the previous step.
5. Add a new run, from start to stop, to the output run-list.
6. Return to step 1.

3.3.1.2 Intersection (rle-and rle1 rle2 ... rle*n*)

This algorithm scans through all of the input run-lists simultaneously moving from left to right. The algorithm is driven by a loop that finds the start column and then the stop column for each run in the output run-list. The loop terminates as soon as it has scanned past the end of any one of the input run-lists.

(**run-list-AND** *rls*)

Find the intersection of the sequence of run-lists *rls*.

1. If any of the run-lists is empty then finish.
2. Set start to be the start column of the rightmost first run in *rls*.
3. Remove each run that is to the left of start, i.e., whose stop column is less than start. If any of the run-lists are empty then finish.
4. If all of the the 1st runs in *rls* overlap then add a new run to the output run-list. In terms of the first run in each run-list: the new run starts at the rightmost start column and ends at the leftmost stop column.
5. Remove the run with the leftmost stop column.
6. Return to 1.

Finding the logical inverse of a run length encoded image is very straightforward: each stop column in the input RLE becomes a start column in the output RLE and each start column in the input RLE becomes a stop column in the output RLE. We create the inverse one run-list at a time:

(**rle-not** rle)

1. Initialize *start* to most-negative-fixnum.
2. If the run-list is empty then set *stop* to most-positive-fixnum otherwise set *stop* to the start column of the first run in the run-list.
3. Output a new run from *start* to *stop*.
4. Set *start* to the stop column of the first run in the run-list
5. Remove the first run in the run-list.
6. Return to step 2.

3.3.2 Envelope

(**rle-envelope** rle radius)

Specifying an algorithm for creating the envelope for a run length encoded image turns out to be straightforward but implementing it is somewhat tricky. We find the envelope one run at a time. The envelope of each run is a rectangle with semicircular ends, to find the complete envelope we find the union of the envelopes for each run in the input RLE. It is not necessary to find the complete envelope for every run, only the portions of each run that are not overlapped by a run on an adjacent row need to be processed.

The implementation of the algorithm is based on two complex operations: run decomposition and run merging. The run decomposition operation visits each run in the input RLE and determines what portions of that run to create a partial envelope for. Each partial run envelope is represented as a new RLE. The run merging operation destructively finds the union of each run in the partial envelope RLE and the output RLE.

3.3.2.1 Run Decomposition

The envelope for an isolated run is a rectangle with a semicircle at each end. The ends are circular arcs with radius equal to the radius of the envelope. The envelope for a run whose length is greater then the radius of the envelope can be represented by a small table.

The only parts of an input run that contribute to the output RLE are the run itself and the portions of the run that are not bordered by an overlapping run on an adjacent row. This is because the adjacent run will always contribute a superset of what its neighbor will. The run decomposition operation visits each run and decomposes it into a set of intervals that are not overlapped by an adjacent run above and another set of intervals that are not overlapped by adjacent runs below. For each interval that does not include either end of the input run we merge N interval length runs with the output RLE. N equals the radius of the envelope, the output runs are stacked vertically above or below the input run. If the interval does include one or both endpoints of the input run then the length of each output run is tapered according to the radius of the envelope. The length of each output run equals the length of the interval extended by the corresponding portion of the semicircle. This value can be looked up in a table like the one shown in Figure 3-9. Figure 3-10 illustrates the run decomposition algorithm.

3.3.2.2 Run Merging

The run merging operation destructively finds the union of a run and an RLE; the real problem is merging the run with a run-list.

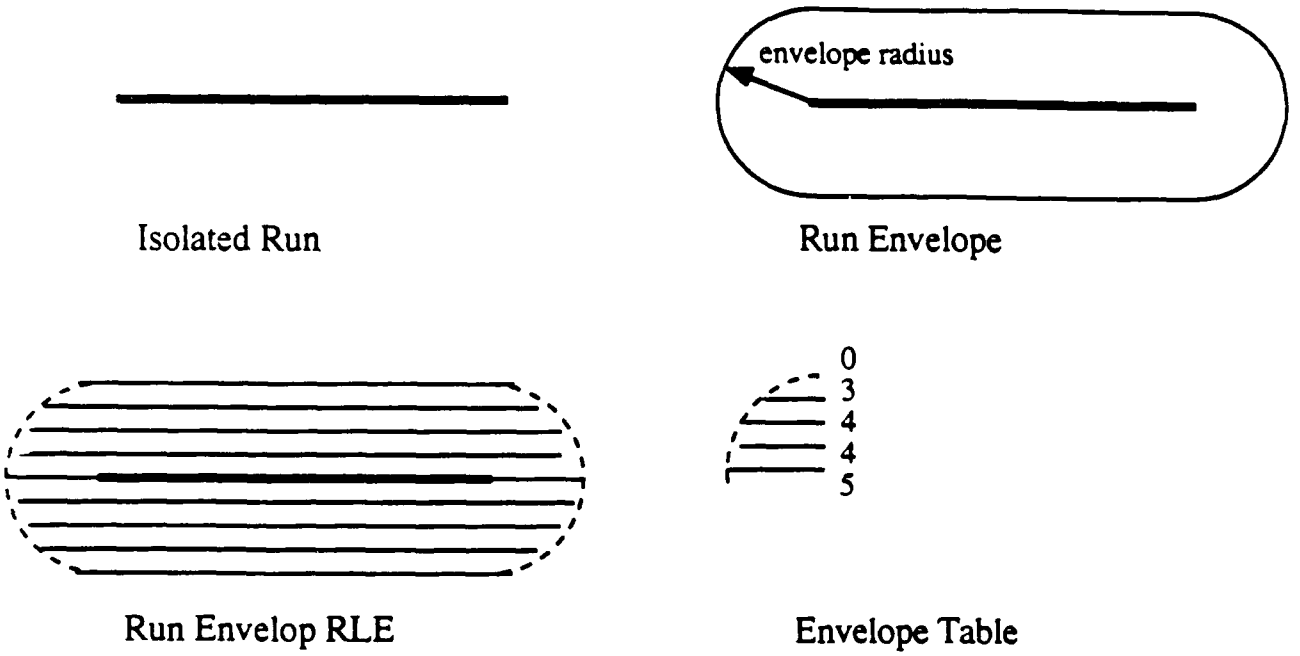


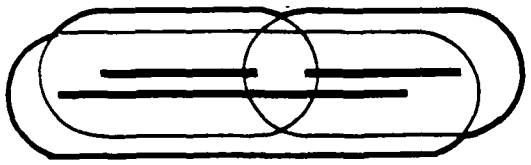
Figure 3-9: Envelope for an Isolated Run

1. First handle the easy cases: There aren't any runs on this row yet or the *run* belongs at the beginning or the very end of this run-list.

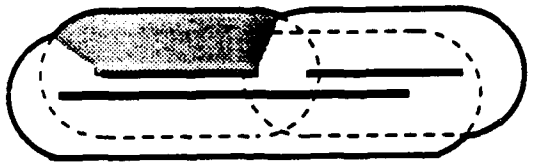
2. Otherwise the *run* belongs somewhere in the middle of the run-list.

   - Scan through the run-list until the start column of the *run* lies in between two successive runs in the run-list.
   - Check for the simple insertion case first (see Figure 3-11).
   - Handle the general case. We know that the run overlaps either the first and/or the second run in the run-list consequently it will not be necessary to insert a new run, it is only necessary to extend the first run overlapping run.
   - If *run* overlaps the first run then remove all of the successive runs that overlap *run* and set the stop column of the first run equal to the stop column of the rightmost overlapped run (see Figure 3-12).
   - If *run* overlaps the second run then remove all of the successive runs that overlap *run* and set the stop column of the second run equal to the stop column of the rightmost overlapped run (see Figure 3-13).

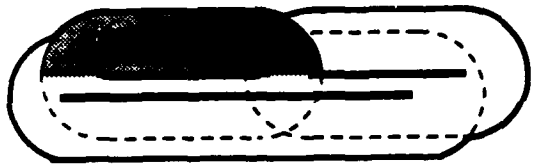
A run length encoded image (RLE) composed of 3 runs.



The envelope for an RLE is equal to the union of the envelopes for each run.

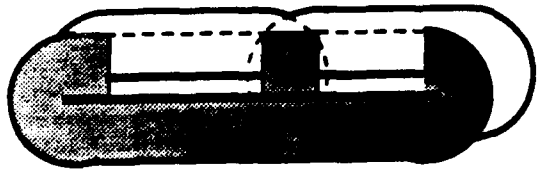


The exact portion of the complete envelope contributed by run 1



Conservative approximation to the portion of the envelope contributed by run 1.

Portion of the envelope contributed by run 2.



Portion of the envelope contributed by run 3.

Figure 3-10: Run Decomposition Example

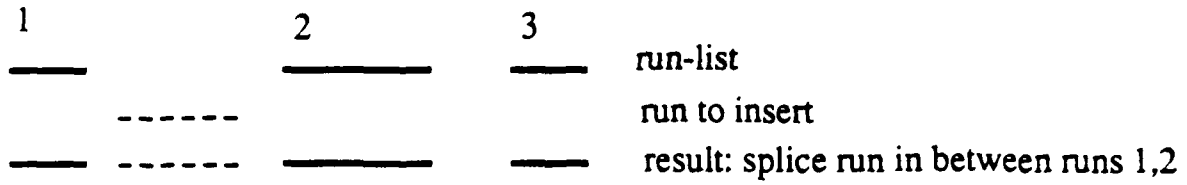


Figure 3-11: Merge Run: Simple Insertion

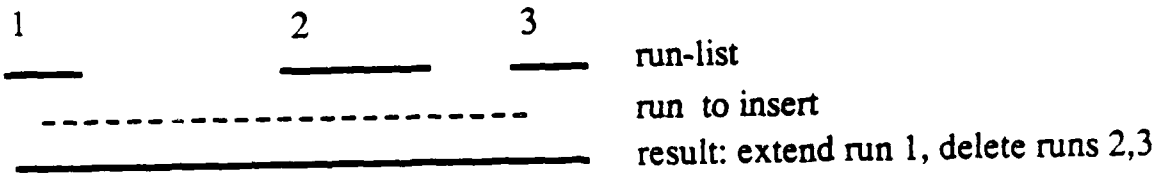


Figure 3-12: Merge Run: Extend the First Run

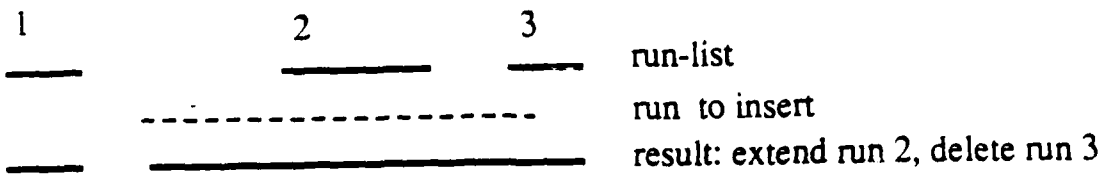


Figure 3-13: Merge Run: Extend the Second Run

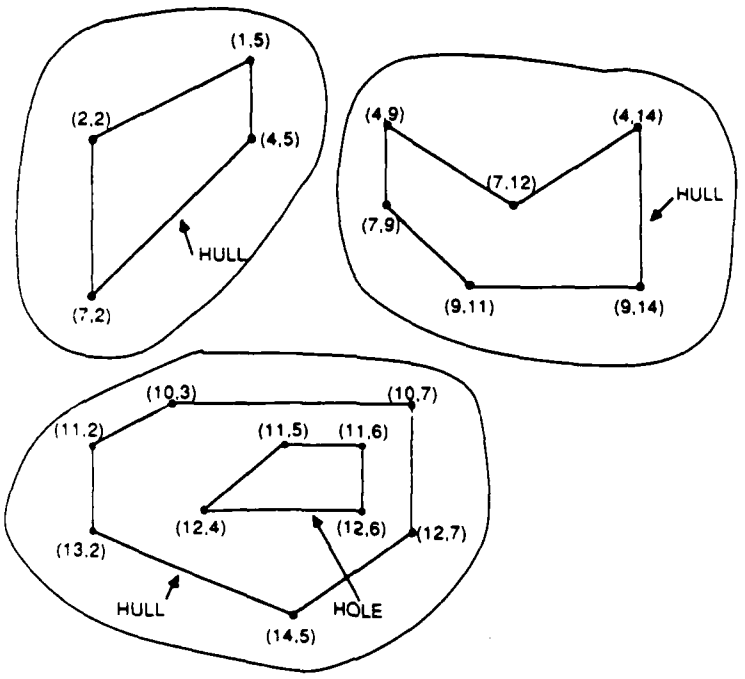


Figure 3-14: Polygon Representation

3.4 Polygon Encoded Image Operations

This section contains a description of the design of some of the algorithms used with a straight line polygonal representation, as well as a description of the associated data structures.

In this research, the basic representation used for polygons is a set of connected straight line segments (Figure 3-14). There is a three-level hierarchy of components from vertices to polygons.

The three levels are:

1. **Vertex**, a pair of coordinates, *row* and *col.*
2. **Edge**, a link consisting of a pair of *vertices* and two pointers, a next pointer pointing to the adjoining *edge* in the clockwise direction and a prev pointer, pointing to the adjoining edge in the counter-clockwise direction.
3. **Polygon**, a closed curve called the *hull* and a number of interior curves called *holes.*

Each of these levels is represented by a Common Lisp structure:

```
;;;2D-VERTEX STRUCTURE:
;;;
(defstruct (2d-vertex (:conc-name vertex-)
                      row
                      col))
```

```
;;;2D-EDGE STRUCTURE
;;;
(defstruct (2d-edge (:conc-name edge-))
           v1            ;type: 2-D vertex
           v2            ;type: 2-D vertex
           next          ;type: 2d-edge
           prev          ;type: 2d-edge
           )


;;;2D-POLYGON STRUCTURE:
;;;
(defstruct (2d-polygon (:conc-name polygon-))
           hull          ;type: 2-D edge
           hole-list     ;list of 2-D edges, 1 per hole
           )
```

3.4.1 Union and Intersection

3.4.1.1 Convex Case

The convexity of the polygons allows us to subdivide the plane they occupy into regions in which the intersection is easily computed [Prep 85]. To subdivide the plane, we choose an arbitrary point P in the plane. From P we draw lines to each of the vertices of n_e and n'_e. We now sort these lines by angular order around P. If P is inside a polygon this sorting is trivially determined by the order of the vertices. If P is outside a polygon, say at infinity, the lines can still be sorted into angular order by walking the vertices of the polygon examining the next and previous vertices, sorting each triple into the correct order.

When the lines have been sorted for each polygon they can be merged into a single ordered list. These lines now divide the plane into sections, the ordering of which constrains the locations of possible intersections. The polygons can be thought of as being sliced into sections by the lines. These sections are, in general, quadrilaterals, but when P is at infinity they are trapezoids (Figure 3-15). The ordering of the lines allows this slicing to be done in time proportional to the total number of edges.

The intersection of the two polygons in a given section is the intersection of the slices of the polygons in that section. The individual intersections can be merged together by a single ordered pass of the sections. If desired, extra colinear vertices, introduced by the slicing procedure, can be removed.

3.4.1.2 Non-Convex Case

The boolean comparison of arbitrary simple polygons is more complex. While still assuming simple polygons, we handle the comparison of sets of disjoint, possibly concave polygons with holes. The algorithm, derived from [Weil 80], merges the

individual polygons being compared into a graph representation where the boundaries of the polygons are arcs in the graph. This graph structure has embedded in it the output polygons of each boolean operation. Hence one application of the algorithm is all that is needed to compute AND, OR, and NOT. Other operations useful in computer graphics, such as clipping, can also be computed with no additional effort.

The polygon boundary representation needs to be extended to allow independent clockwise or counter clockwise traversal of an edge. The resulting winged edge structure (Figure 3-16) has vertices v1 and v2 where v1 is the first vertex encountered in a clockwise traversal of the original polygon.

This can be represented by a LISP structures as follows:

```
;;;
;;; An Edge-Side is a basic 2D-edge with the addition
;;; of a HISTORY which is a list of regions associated
;;; with the edge.  The edge-side may also have an
;;; ENTRY-POINT is T if the edge-side is the entry
;;; point of a contour.
;;;
(defstruct (edge-side (:include edge)
                      (:conc-name edge-))
  parent
  history
  side
  entry-point)

;;;
;;; A WINGED-EDGE has two vertices (with a convention about their
;;; usage), two lists of regions (for regions on either side of the
;;; edge), and four pointers (two for each side pointing to the
;;; next an previous).
;;;
;;; V1: first vertex past when traversing side1 CW
;;;     last vertex past when traversing side2 CW
;;; V2: first CW vertex for side2
;;;     first CCW vertex for side1
;;;
(defstruct (winged-edge (:include edge)
                        (:conc-name edge-))
  deleted  ; t if it is a removed coincident edge - allows traversal
           ; of original graph - will not be intersected with
           ; anything
  parent-curve
  subject  ; t if the edge is in the subject poly, nil if in clip
  side1    ; edge-sides point to "edge-side" structures
  side2)
```

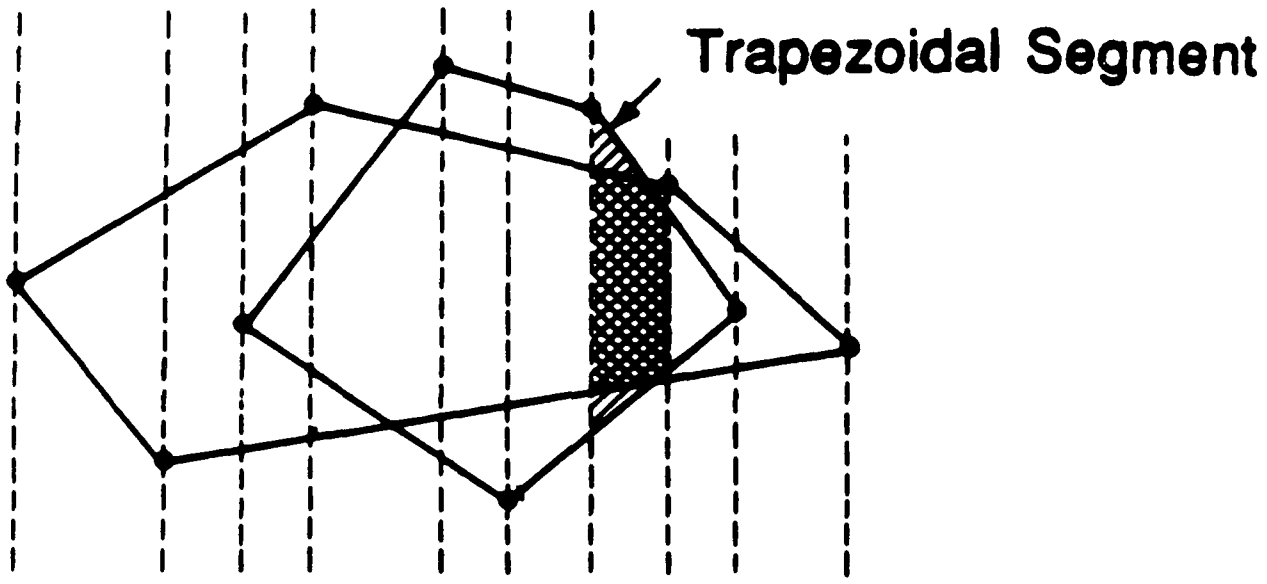


Figure 3-15: Segments defined by vertices with P at infinity.

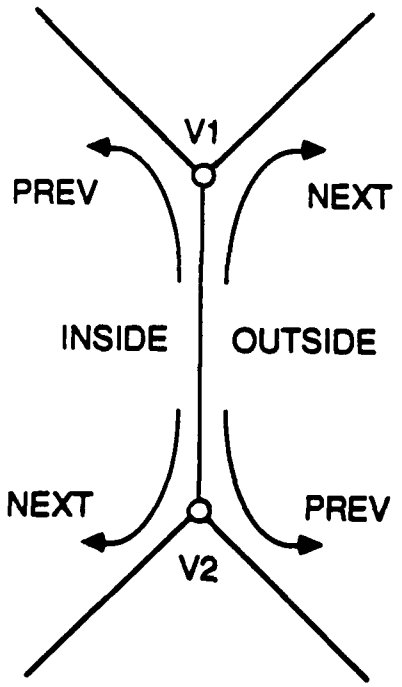


Figure 3-16: Winged Edge data structure.

The structure has two sides; side1 is inside the polygon and side2 is outside. Each of the edge sides has its own next and previous pointers to edge sides encountered in a clockwise or counter clockwise traversal respectively. Edge sides are thought of as facing the area they bound. Associated with each of these edge sides is a list of regions that it faces.

The algorithm has four stages:

1. spatial analysis of input,
2. graph building,
3. traversal,
4. spatial analysis of output.

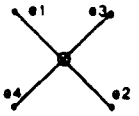
The first step is to establish the topological relations between the input polygons. If any of the contours of one of the input polygons is completely contained within the contour of another polygon then the histories of the edge sides of the contained contour must be updated to reflect the new region these sides face.

The main component of the graph building stage is a graph-merge process which determines intersection relationships between edges of the polygons and merges the boundary representations into the graph structure in a way which reflects the type of intersection. Edges in the graphs are assumed to be straight, though the algorithm could be modified to deal with curved edges. There are essentially three types of intersections:

- midpoint-midpoint - Two edges cross each other. The edges are split at the intersection and merged into the graph (Figure 3-17).
- endpoint-midpoint - The endpoint of one edge is incident on another edge. The intersected edge is split and the intersecting edge is added to the graph (Figure 3-18). Additionally, all edges incident on the vertex of the intersecting edge must be added to the graph using the endpoint-endpoint merge.
- endpoint-endpoint - The vertices of two edges coincide. The edge and all edges incident on the intersecting vertex of that edge are merged into the graph (Figure 3-19).

The merge process uses information about whether the intersecting edges are inside or outside the contour being intersected and whether the edge is entering or leaving the contour. In endpoint-endpoint merge the spatial relationships of the surrounding edges are used to merge the edge consistently.

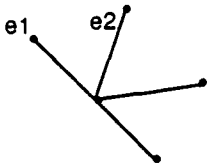
An additional step is necessary to detect and properly handle coincident edges. When an edge which has been merged into the graph is detected to be coincident one of the two coincident edges is deleted, the pointers are updated, and the edge side history information of the deleted edge is incorporated into the histories of the remaining edge sides.



- SPLIT BOTH EDGES AT INTERSECTION
- DETERMINE IF e3 IS INSIDE OR OUTSIDE CONTOUR OF e1 - e2
- DETERMINE IF e3 IS ENTERING OR EXITING
- UPDATE POINTERS

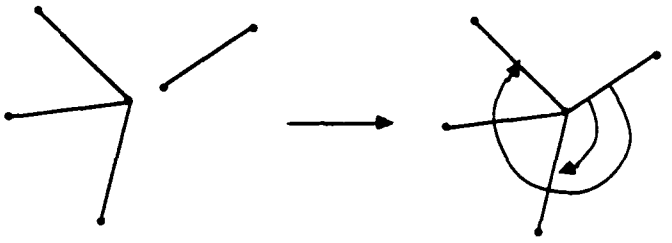
- SIMPLE AND EFFICIENT

Figure 3-17: Midpoint-Midpoint Intersection



- END POINT OF e2 LIES ON e1
- SPLIT e1 AND MERGE e2 DOING ESSENTIALLY THE SAME THING AS MID-MID CASE
- FOR ALL EDGES INCIDENT ON THE MERGED VERTEX, MERGE USING END-END

Figure 3-18: Endpoint-Midpoint Intersection



- FIND INCIDENT EDGES WITH MIN AND MAX ANGLE FROM MERGING EDGE
- FIND THE CONTOUR COMMON TO THE MIN AND MAX EDGES

Figure 3-19: Endpoint-Endpoint Intersection

When all edges have been merged, the graph contains all the intersection information necessary for performing any of the set operations. Thus the graph need only be generated once for a given pair of polygons. The set operations then resemble queries to this structure.

Each initial contour is said to have two entry points; one for each side. The entry points provide a starting place from which to traverse a contour. When edges are intersected they may introduce new contours which have no entry point. Hence, every intersection generates a new entry point. The traversal process walks along the edges of all the contours in the new graph by starting at an entry point and following it until it forms a closed loop. Redundant entry points formed in the merge process are eliminated during traversal.

From the contours in the new graph the contours matching the desired boolean operation can be selected. The appropriate contours for a given operation are determined by list of regions which they face.

- OR -- collect contours with no history information and hence, inclose no region.
- AND -- collect contours which inclose the regions of both input polygons.

The spatial analysis of the output contours determines the spatial relationships of these new contours. The contours are sorted into a list of disjoint polygons with holes. The result is a generalized polygon in boundary representation.

3.4.2 Inverse

In the case of Grids and RLE's the inverse was specified with respect to the fixed grid on which these representations are defined. This grid defines an implicit finite universe which limits the spatial extent of the inverted representation. In the case of a polygonal representation, no such bounding universe exists *a priori*. We could assume that the inverse of a polygon has no explicit boundary; that is, it extends to infinity. This however, is not a practical solution.

The inverse could also be defined with respect to any arbitrarily shaped polygon. In this case, the inverse can be specified as one of the boolean set operations described above. The result is equivalent to *clipping* the subject polygon with the arbitrary universe polygon. This operation is more general than that discussed for Grids or RLE's. To be consistent with the grid and RLE inverse operation, the universe polygon is considered to be the polygonal boundary of the grid used as the universe for these other representations.

A special case of polygon inversion which exists when the universe is known to completely contain the subject polygons. In this case the universe forms the hull of the new polygon. This hull has holes which are the hulls of all the subject polygons being negated. Additional disjoint polygons are formed by the holes of the subject polygons. In this case, no intersection operations are necessary, and all spatial relationships are already known.

3.4.3 Envelope

The following in a simple algorithm which works on both convex and non-convex polygons. The simple algorithm ENVELOPE below makes use of the UNION operation.

1. We construct a box around each line segment of (2 * envelope radius) in width by length of the segment.

2. We also construct a straight-line approximation of a circle using the enveloping radius around each vertex.

3. Union all the polygons constructed in steps 1 and 2.

The performance of the algorithm can be enhanced by using a special-purpose union operation that is optimized for the intersection of rectangular boxes.

More elegant enveloping algorithms follow.

3.4.3.1 Convex Case

We will assume that the enveloping algorithm is performed on a convex polygon. The following algorithm relies on the fact that there is a one to one mapping between vertices of the original convex polygon and those of the enveloped polygon.

The algorithm is quite simple. For each vertex of the original polygon, the corresponding vertex of the enveloped (returned) polygon is determined as follows. Let the coordinates of the present vertex be (u_2, v_2), those of the previous vertex be (u_1, v_1), and those of the next vertex be (u_3, v_3), where "previous" and "next" are defined with respect to a clockwise traversal of the polygonal boundary. Then, the coordinates (u_2^e, v_2^e) of the corresponding vertex of the enveloped polygon are given by (see Figure 3-20)

$$u_2^e = \frac{1}{m_1 - m_2}\left(u_2[1+m_1] - v_2[1+m_2] - RS_1\sqrt{1+m_1^2} - 2RS_2\frac{m_2}{\sqrt{1+m_2^2}}\right)$$

$$v_2^e = \frac{1}{m_1 - m_2}\left(u_2 m_1[1+m_2] - v_2 m_2[1+m_1] - RS_1 m_2\sqrt{1+m_1^2} - 2RS_2\frac{m_1 m_2}{\sqrt{1+m_2^2}}\right),$$

where

$$\begin{aligned} R &\equiv \text{enveloping radius} \\ m_1 &\equiv (v_2 - v_1)/(u_2 - u_1), \\ m_2 &\equiv (v_3 - v_2)/(u_3 - u_2), \\ S_1 &\equiv (v_2 - v_1)/|v_2 - v_1|, \\ S_2 &\equiv (v_3 - v_2)/|v_3 - v_2|. \end{aligned}$$

Only a single traversal of the polygon is required to compute these quantities.

From \ To	Grid	RLE	Polygon
Grid		✓	✓
RLE	✓		✓
Polygon	✓	✓	

Table 3-1: Update Methods

3.4.3.2 Non-Convex Case

In the case on non-convex polygons, enveloping becomes more complicated. Following the technique described above can lead to non-simple polygons; that is, polygons with overlapping edges (Figure 3-21).

An algorithm for the non-convex case proceeds in two stages. The first stage performs the algorithm described for convex polygons. The second phase transforms potentially non-simple polygons into simple ones. This second phase conceptually corresponds to performing a union of the polygon with itself. This makes the algorithm more complex and computationally expensive.

3.5 Update Algorithms

Given data in a representation R and a method M there are possibly many ways to compute M. For example an algorithm M_R which uses representation R might be used. This is straightforward, but it may be the case that M_R is computationally expensive. There may be some faster algorithm M_X which implements the method but relies on having data in the representation X. If the combined cost of converting the data from R to X, performing M_X, and possibly converting the result back to representation R, is less than the time to compute M_R directly from R, then this more complex path may be desirable.

To permit this type of opportunistic use of different representations of the same data to optimize processing, conversions must exist from one representation to another. A complete matrix of *update* operations allows any representation to be directly converted to any other. The complete matrix has been implemented for Grids, RLE's and Polygons (Table 3-1). The algorithms themselves are described in the following sections.

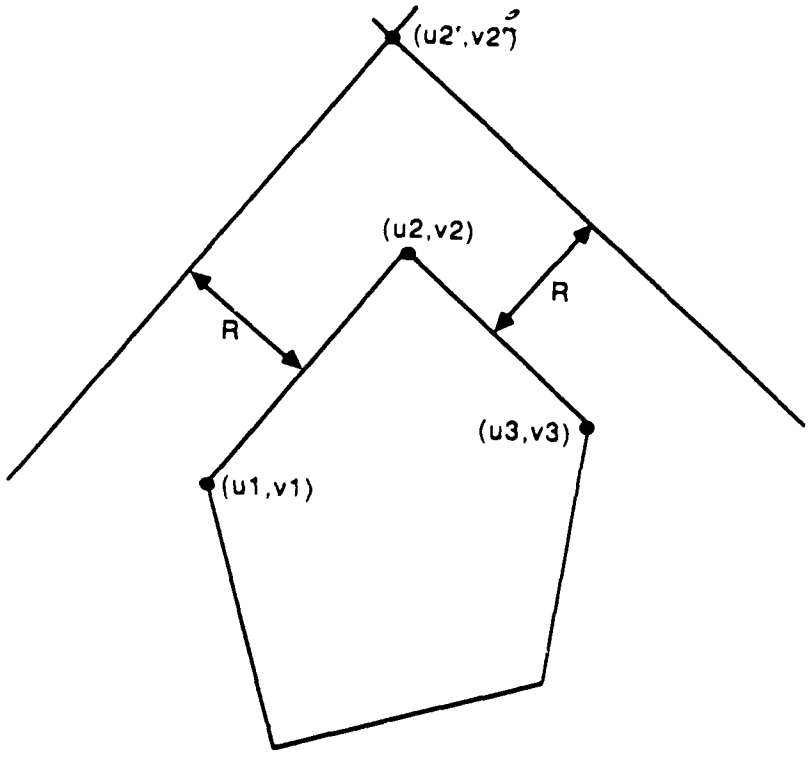


Figure 3-20: Definition of terms used in enveloping of convex polygons.

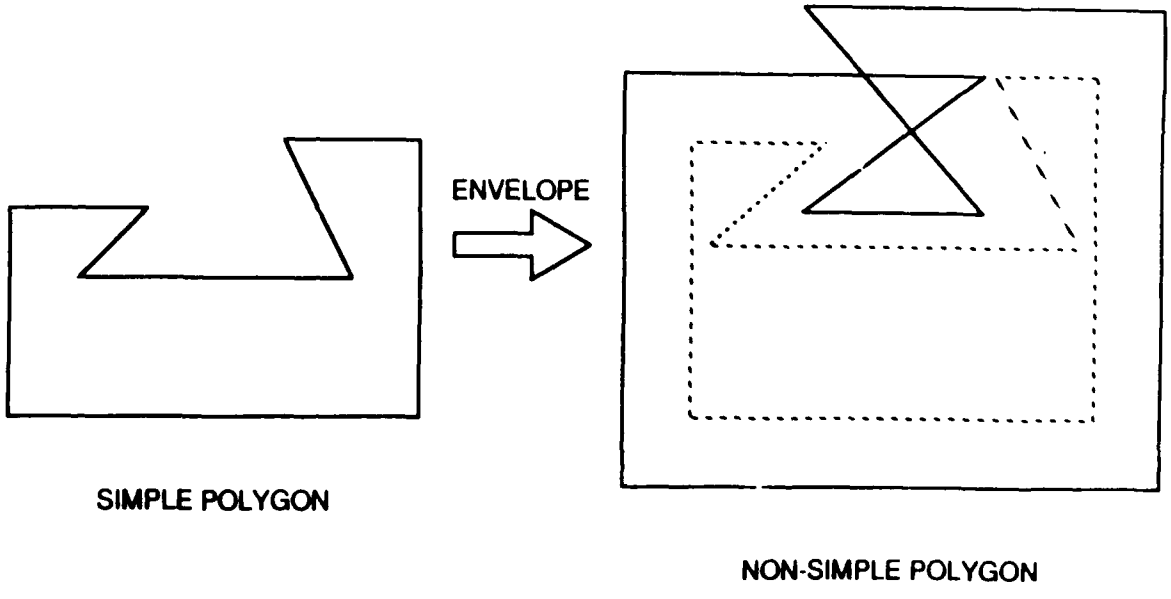


Figure 3-21: Enveloping of a Non-Convex Polygon which results in a Non-Simple Polygon.

3.5.1 Grid

3.5.1.1 Grid To RLE

The conversion from grids to RLE's is quite straightforward since the RLE is simply a compact encoding of the same information already present in the grid. The algorithm proceeds as follows:

```
for each row in the grid
  run-list <- ()
  start <- ()
  for each grid cell
    if grid cell = 1 and start = () then
       start <- current column
    if grid cell = 1 and (not start = ()) then
       nothing
    if grid cell = 0 and start = () then
       nothing
    if grid cell = 0 and (not start = ()) then
       append the pair (start . current column) to the
         end of run-list
       start <- ()
  if (not run-list = ())
    add run-list to rle
```

3.5.1.2 Grid To Polygon

Converting a grid representation to a polygonal one is more difficult than converting from grid to RLE. The reason is that the two representations encode different information; the polygon encodes the object boundary which is not explicitly represented in grid format.

The purely binary grid contains no information about the connectedness of regions. Hence, the first step in converting grid to polygon, is to form a connected components image where each disjoint region in the grid is assigned a unique label. An *8-connected* image can be created with a 3x3 mask. When the mask is centered on a non-zero grid cell, it is 8-connected to all the other non-zero cells in the mask. These cells are labeled consistently in the connected components image.

Once the regions have been isolated, their boundaries can be extracted. The cells of the grid are examined until a new label is found. The boundary of the region is then followed by examining the 8-connected neighbors of that cell and the direction of movement to each non-zero cell. The cell with the path whose directions is most counter clockwise with respect to the previous direction is chosen as the new cell. The process continues until the boundary is traversed. The interior of the region must then be scanned for holes.

The boundary of a grid-based region is not smooth since it is a discritization of a possibly continuous curve. For example a diagonal line, when represented in a

grid, takes on a jagged appearance. Hence, it is necessary to smooth the extracted boundary by fitting straight line segments to jagged contours.

3.5.2 RLE

3.5.2.1 RLE to Grid

As with the case of Grid to RLE, RLE to Grid is straightforward. A grid is created that has the same dimensions as the implicit grid underlying the RLE. The grid cells are initialized to be all 0's. Each run-list of the RLE is examined, and for each run in the list 1's are written into the grid cell corresponding to that row and the columns corresponding to the run.

3.5.2.2 RLE to Polygon

The conversion of an RLE to a Polygon uses the same notion of 8-connectivity as used in Grid to Polygon. In this case, however, the algorithm moves from run-list to run-list examining the runs to determine connectivity. By reasoning about the overlap of runs not every pixel represented by the run need be examined.

3.5.3 Polygon

3.5.3.1 Polygon to Grid

The conversion from polygon to grid is a standard *scan-conversion* problem common to raster graphics displays. Due to interest in graphics displays, and the requirement that such displays be fast, much attention has been paid to the scan-conversion problem [Fole 82].

The algorithm produces *scan-lines* which correspond to rows in the output grid. Each line is intersected with the non-horizontal edges of the polygon to produce a list of intersections. The grid cells on the scan line which lie between these intersections and fall inside the polygon must be determined. The portions of the scan-line which lie inside the polygon can be determined by counting from left to right the number of intersections of the line with the polygon. When the count is odd the line is inside the polygon, and when it is even the count is outside the polygon. Horizontal edges can be ignored, entirely. The only difficulty occurs when a scan-line intersects the polygon at a vertex. A check for vertex intersections deals with this special case.

Where the scan-line is completely within the polygon, corresponding grid cells are easily determined. At the polygon edges, where the scan-line enters and leaves the polygon, determining whether or not a grid cell is inside or outside the polygon is more difficult. We rely on *Bresenham's line algorithm* [Bres 65] to determine the correct grid location of the polygon edges for a given scan-line. This algorithm chooses the best grid locations for representing the edge, and does so quickly by using only integer arithmetic.

3.5.3.2 Polygon to RLE

The polygon to RLE algorithm is almost identical to the algorithm for polygon to grid. The difference being, that the interior points do not need to be calculated. All that is needed are the Bresenham generated intersection points, and the knowledge of where the scan-line enters and exits the polygon. From this information, a run-list is calculated for each scan line, with the start and stop points of each run being generated by the Bresenham algorithm. The RLE produced will be identical to the polygon only if the polygon is only composed of vertical and horizontal edges. If the polygon has diagonal edges, then the similarity will be limited by the accuracy of the Bresenham routine.

4. Performance Study

4.1 Structure of Study

The work of the previous Chapter provides the foundation for the performance studies of this Chapter. The performance study is divided into the following stages:

- analysis of the complexity of the algorithms under study,
- development and evaluation of a inherent data complexity measure,
- evaluation of algorithm performance,
- generation of performance models.

The results of this study will show how to build a system which selects an efficient processing sequence of algorithms and representations. It will also indicate the viability of using a representation independent measure of data complexity to generate performance models for algorithms. If this is possible, the study itself becomes a model for generating such performance models.

4.2 Complexity of Algorithms

In this section we discuss the computational complexity of enveloping, and boolean set operation algorithms for images represented by grids, polygonal boundaries, and run-length encodings (RLE's). We also examine the complexity update algorithms for converting among grid, RLE, and polygonal data structures.

The enveloping algorithm takes an image object and returns a new object which is comprised of the original one plus all points that are within a fixed (Euclidean or Manhattan) distance from the object.

The boolean set operations being considered are OR, AND, NOT. AND and OR return the intersection and union of two image objects; and NOT returns the complement of an object.

A summary of our results are shown in Figure 4-1.

4.2.1 Operations on Grid Encoded Images

The locations of objects in images which are represented by either polygonal boundaries or run-length encodings are explicit in the representation. In contrast, if we wish to perform algorithms on the objects in a pixel array or grid representation, we must first locate the objects in the image. The location information required depends on the particular algorithm to be performed. For example, adjacency algorithms require that the boundary of an object be known, whereas boolean set operations require that all points in the objects be known.

For this discussion, grid algorithm complexity is divided into two parts. First we discuss the complexity of the operations used to obtain the appropriate object location

			Polygons	
	Grids	RLE's	Convex	Non-convex
Enveloping	$(2R+1)^2 n_o$	$R n_{rl}\, n_r$	n_e	n_e^2
OR	N	$n_r n_{rl} + n_r' n_{rl}'$	$n_e + n_e'$	$n_e\, n_e'$
AND	N	$(n_r + n_r')\min(n_{rl}, n_{rl}')$	$n_e + n_e'$	$n_e\, n_e'$
NOT	N	$n_{rl}\, n_r$	$n_e + u_e$	$n_e + u_e$

Notation: R = enveloping radius
Grids: n_o = no. of points in object
d = typical object diameter; N = no. of image pixels
RLE's: n_{rl} = no. of run lists; n_r = average no. of runs per list
Polygons: n_e = no. of edges
u_e = no. of edges in universe

Table 4-1: Summary of Processing Algorithm Complexity Results

information. Secondly, we discuss the complexity of performing the algorithm on the image object with known location. Since in practice, varying degrees of initial location information are available, we further break down the process of locating the object into several steps. In order of increasing information content (and, hence, increasing algorithmic complexity), these steps include knowledge of:

1. No points in the object,
2. 1 point in the object,
3. All points in the object,
4. All boundary points, and
5. Boundary points and bounding box.

We shall first give the complexity of arriving at each of these informational states. In our discussion of specific algorithms, we will specify the state of knowledge that we are assuming as a starting point. The computational overhead involved in starting from a different level of knowledge can simply be added on.

Below is a list of algorithms and their computational complexity for increasing the degree of object location knowledge as specified at the beginning of each entry. We assume that we are given an N-pixel image which contains a single object comprised of n_o pixels, n_b of which are on its boundary. We will let d signify a typical diameter of the object.

- No points $\rightarrow$ 1 point. The straightforward approach is to simply go through the image line by line until a point in the object is located. In the worst case, this would take $O(N-n_o)$ steps. If it is known that the object is fairly compact (that is, not stringy), then a more efficient approach is to sample the image at intervals of approximately d pixels. The number of such intervals, and thus the complexity of this algorithm, is $O(N/d^2)$.

- 1 point $\rightarrow$ all points. Given one point in an image object which is known to be convex, a region growing algorithm can be used to locate all remaining points in the object (see Figure 4-1). The figure illustrates a region-growing algorithm for 4-connected convex image objects. The initial image object point corresponds to the central grid cell in this diagram. From the initial point, new pixels are tested for membership in the object in a radial pattern as shown. The precise pattern used to expand from the initial point is indicated in the figure by the progression from heavy to light lines. Any point which is found to lie outside the image is not expanded. The image object thus consists of all pixels which have been expanded. This algorithm checks each point only once and stops on the boundary and is thus non-redundant. This requires $O(n_o)$ time. This algorithm is insufficient for image objects of arbitrary shape, however. For example, it will not find all points in an object with overhangs. In general, an algorithm which determines the set of connected components in the entire image or within some bounding box of the image object must be used. Such an algorithm requires a first pass through the image or bounding box to establish initial pixel labels and to build up a list of pointers which is used in a second pass to determine the correct pixel labels from the initial values.

- One point or all points $\rightarrow$ all boundary points. Given all points, a brute force $O(n_o)$ approach would be to simply check them all and store those on the boundary. For an object of arbitrary topology, that is, one that may have holes, corresponding to multiple boundaries, this is a reasonable approach. More efficient approaches are possible if the object is known to have no holes. Then, starting from any point in the object, we need only to first locate a point on the boundary ($O(d)$) and then follow the boundary around the object ($O(n_b)$).

- Boundary points $\rightarrow$ bounding box. The simplest approach is to go through the list of all boundary points to determine the smallest and largest coordinates of the object in each direction. This is obviously an $O(n_b)$ operation. If the boundary points are listed in order corresponding to traversal of the boundary, then computational effort may be reduced by using the fact that the coordinates of consecutive boundary points differ from each other by no more than 1 in each direction. Thus, if the coordinates of the boundary point being examined are d away from the minimum and maximum boundary coordinates found thus far for each direction, then the d subsequent points in the list can be safely ignored.

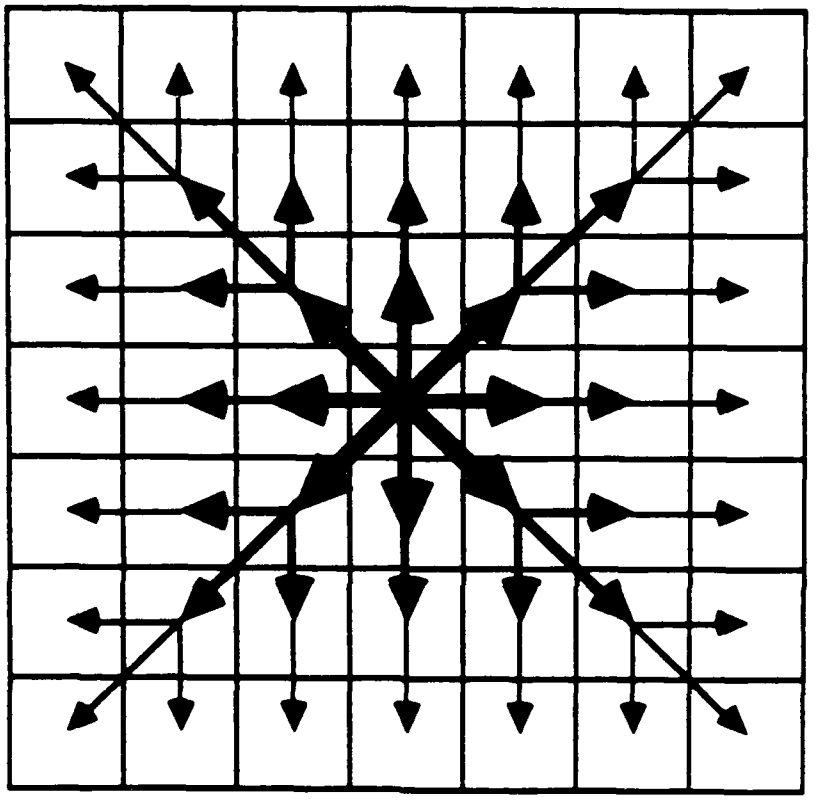


Figure 4-1: Region-growing algorithm for 4-connected convex image objects.

4.2.1.1 Enveloping

The enveloping algorithm that we now discuss for grid-based images returns an image object which consists of the original image object and all pixels within a given Manhattan distance R from it. We assume that the image contains a single object whose boundary and bounding box are known.

The algorithm computes the chamfer array [Barr 78] within an expanded bounding box. The bounding box is specified by its minimum and maximum x and y coordinates. We expand the box outwards by R in each direction by letting

$$\begin{aligned} x_{\min} &\leftarrow x_{\min} - R \\ x_{\max} &\leftarrow x_{\max} + R \\ y_{\min} &\leftarrow y_{\min} - R \\ y_{\max} &\leftarrow y_{\max} + R. \end{aligned}$$

It is easily seen that the image returned by enveloping will contain points on the boundary of, but not outside, this enlarged bounding box. Therefore, we compute the chamfer array within the enlarged bounding box. This algorithm gives the Manhattan distance of each cell from the image and requires only two passes through the box. The enveloped (returned) image contains all pixels less than or equal to R away from the image (including the image itself, which is a distance of 0 away). Since the enlarged box contains on the order of $(2R+d)^2$ pixels, this algorithm is $O((2R+d)^2)$.

A more straightforward algorithm involves passing a mask of size $2R+1$ by $2R+1$ over the grid. Each grid cell is examined, and if the cell contains a 1 then each cell covered by the mask, centered on that cell, is set to 1. Since all cells are examined, the cost is $O(N)$. The actual running time is dependent on the number of grid cells that are 1: n_o.

If a connected components representation is available, an enveloping algorithm can be written which traces the boundary points of the labeled regions with the same mask as above. The algorithm still has complexity $O(N)$ but will involve setting fewer grid cells to 1's, and hence may be faster.

4.2.1.2 Boolean Set Operations

The boolean set operations (union and intersection) for grids are straightforward. They involve looking at each grid cell in each of the input grids. Since the spatial correspondence between input grids is known, each cell is compared with only one other cell. The algorithms then examine $2N$ grid cells and set N grid cells in the output grid where N is the total number of cells in the grid. The resulting complexity is $O(N)$.

4.2.1.3 Negation

The negation operation is similar to the other boolean operations. Each grid cell is examined and its value is switched from 1 to 0 or 0 to 1. This involves a cost of $O(N)$.

4.2.2 Operations on Polygon Encoded Images

The data structure we use for polygon encoded image objects is a list of 4-tuples, one for each edge. Each 4-tuple contains the coordinates of vertex 1 and vertex 2 (where the vertices of an edge are numbered in the order in which they would be encountered in a clockwise traversal of the polygonal boundary), a pointer to the previous edge, and a pointer to the next edge (where again, previous and next refer to a clockwise traversal). The distinction between inside and outside of the polygon is unambiguous in this representation. Standing at vertex 1 of an edge and looking toward vertex 2, the inside of the polygon is to the right and the outside is to the left. We will label the number of polygon edges (also the number of vertices) by n_e.

4.2.2.1 Enveloping

Convex Case We will assume that the enveloping algorithm is performed on a convex polygon. The algorithm relies on the fact that there is a one to one mapping between vertices of the original convex polygon and those of the enveloped polygon.

The algorithm is quite simple. For each vertex of the original polygon, the corresponding vertex of the enveloped (returned) polygon is determined as specified in Section 3.4.3. Since only a single traversal of the polygon is required to compute these quantities, this algorithm is $O(n_e)$.

Non-Convex Case In the case on non-convex polygons, enveloping becomes more complicated. Following the technique described above can lead to non-simple polygons; that is, polygons with overlapping edges (Figure 3-21).

An algorithm for the non-convex case proceeds in two stages. The first stage performs the algorithm described for convex polygons. The second phase transforms potentially non-simple polygons into simple ones. This second phase conceptually corresponds to performing a union of the polygon with itself. Boolean set operations for non-convex polygons will be described below, and as we will see, boolean operations on polygons with n_e and n'_e edges is $O(n_e n'_e)$. Hence, the union of a polygon of n edges with itself is $O(n^2)$. This increases the complexity of enveloping from $O(n)$ to $O(n^2)$ in the non-convex case.

4.2.2.2 Boolean Set Operations

Convex Case The intersection of convex polygons n and n', with n_e and n'_e edges respectively, can be found in time $O(n_e + n'_e)$. The convexity of the polygons allows us to subdivide the plane they occupy into regions in which the intersection is easily computed [Prep 85].

To subdivide the plane, we choose an arbitrary point P in the plane. From P we draw lines to each of the vertices of n_e and n'_e. We now sort these lines by angular order around P. If P is inside a polygon this sorting is trivially determined by the order of the vertices. If P is outside a polygon, say at infinity, the lines can still be sorted into angular order in time $O(n_e)$. This is done by walking the vertices of the

polygon examining the next and previous vertices, sorting each triple into the correct order.

When the lines have been sorted for each polygon they can be merged into a single ordered list in time linear in the total number of vertices. These lines now divide the plane into sections, the ordering of which constrains the locations of possible intersections. The polygons can be thought of as being sliced into sections by the lines. These sections are, in general, quadrilaterals, but when P is at infinity they are trapezoids (Figure 3-15). The ordering of the lines allows this slicing to be done in time proportional to the total number of edges.

The intersection of the two polygons in a given section is the intersection of the slices of the polygons in that section. That is the intersection is the intersection of quadrilaterals which can be found in constant time. The individual intersections can be merged together by a single ordered pass of the sections. If desired, extra colinear vertices, introduced by the slicing procedure, can be removed in linear time.

Non-Convex Case The boolean comparison of arbitrary simple polygons is more complex. While still assuming simple polygons, we handle the comparison of sets of disjoint, possibly concave polygons with holes. The algorithm, derived from [Weil 80], merges the individual polygons being compared into a graph representation where the boundaries of the polygons are arcs in the graph (see Section 3.4.1.2). This graph structure has embedded in it the output polygons of each boolean operation. Hence one application of the algorithm is all that is needed to compute AND, OR, and NOT. Other operations useful in computer graphics, such as clipping, can also be computed with no additional effort.

The computational expense of the algorithm lies in the intersection testing and the sorting of spatial relationships. In the worst case every edge of one polygon must be tested for intersection with every edge of the other polygon. So for polygons n and n' with n_e and n'_e edges respectively, we have a worst case cost of $O(n_e n'_e)$. In practice, many edges can be eliminate from consideration by simple boxing tests. An initial screening phase, with cost $O(n_e + n'_e)$, can eliminate all edges which do not fall within the minimum bounding rectangle of the other polygon. If the polygons have many more edges than points of intersection the cost of this filtering will likely pay off.

In fact, a hierarchy of boxing tests can be applied to reduce the cost, but not the inherent complexity, of the algorithm. For example, a quick test to see if the minimum bounding rectangles of two edges intersect, can reduce the number of actual edge intersection tests that must be performed.

4.2.2.3 Negation

The complexity of polygon negation varies with the nature of the universe which provides the frame of reference for the negation. In the general case of negating a polygon with respect to an arbitrary universe, negation is treated as another polygon set operation. Consequently, the cost is $O(n_e u_e)$ where u_e is the number of edges in

the universe polygon. In the special case where the universe is a rectangle the cost is $4n_e$ or $O(n_e)$.

When the universe and the polygon being negated are disjoint, the problem more simple. A shuffling of contours achieves the desired result in time $O(n_e)$, since every edge must be copied in the shuffle.

4.2.3 Operations on Run Length Encoded Images

A run length encoded image object consists of a list of run lists, each of which corresponds to a row of the object, where row is defined with respect to the underlying pixel representation. A run list contains the row number and a list of start and stop columns which delimit runs of the object. We will let n_{rl} be the number of run lists in a run length encoded object. The average number of runs in a run list will be denoted n_r.

4.2.3.1 Enveloping

Enveloping of a run-length encoded image may be accomplished in two steps. First, each run of the image is enveloped separately. Second, a run merging operation destructively finds the union of each run in the envelope RLE and the output RLE. A more efficient algorithm, in which only partial envelopes are created for each run, is specified in detail in the Final Technical Report for SDS I. We expect however that the computational complexity of these two algorithms differs only by a constant factor, since on average a partial envelope will contain as many runs as a complete envelope.

If R is the enveloping radius, then to envelope a given run we generate R new runs and destroy the original one. Since there are $n_{rl}n_r$ runs, this requires $Rn_{rl}n_r$ operations. To merge these enveloped runs, we first merge the envelopes of all runs in each row separately. For this we use the algorithm below for OR, the operator which takes the union of sets. The run envelopes for runs in a given row are already aligned. Therefore, this requires only $O(Rn_r)$ operations. Finally, we merge the merged run lists of successive rows. The envelopes of runs in successive rows overlap in at most $R/2$ rows. For the original image object to have been connected, it must be the case that the union of the enveloped runs from successive rows contain no more than the maximum number of runs in the two rows. If, on average, this number is n_r, then the complexity of merging the run envelopes for successive rows is $O(Rn_r)$. Clearly, the greatest computational effort goes into the initial enveloping of each run. Thus, the computational complexity is overall of $O(Rn_{rl}n_r)$.

4.2.3.2 Boolean Set Operations

For the set operations AND and OR we first perform alignment and sorting routines. First we align the two run length encoded image objects by rows. We assume that the run lists are given in order of increasing row number. To line up two image objects we first determine whether their row numbers intersect by comparing

the first and last row numbers of the two objects. The number of integer operations required for this is at most 4. Then placing the rows in correspondence is linear in the number of run lists n_{rl}, n'_{rl} of the two objects. Once they are in registration, each row containing both objects is considered separately. The run lists for the two objects in a given row are sorted based on the first element in each run. We then go through the sorted list, comparing the end column of the ith run in the list with the start column of the $(i+1)$st. In the worst case, comparing run lists in this way requires $O((n_r + n'_r) \times \min(n_{rl}, n'_{rl}))$ time.

For AND we, in addition, label each run by the object it belongs to. For each row that contains runs of both objects, we do the following.

OR Beginning with the first run in the ordered list, and for each run i in order, until the run is copied over to the list of runs in the union, do:
BEGIN
If the start column of run $i+1$ is more than 1 greater than the end column of run i, then copy run i onto the list of runs forming the union for this row.
Otherwise, replace the ith run in the list by a run whose start column is that of the ith run and whose end column is the greater of the end columns for runs i and $i+1$. Eliminate run $i+1$ from the ordered list and renumber the remaining runs.
END
In addition, all run lists for rows which are occupied by a single object only must be copied over to the list of runs in the union as well.

In addition to the computational burden of comparing and sorting the run-length encoded images (see above), each run of both images is either merged to form another run or copied directly onto the list of runs in the union. Thus the additional computational complexity is simply on the order of the total number of runs of both images, that is, $O(n_r n_{rl} + n'_r n'_{rl})$.

AND Compare all pairs of adjacent runs in the list in order. If run i and run $i+1$ are from different objects, and if the start column for run $i+1$ is less than or equal to the end column for run i, then add to the list of runs in the intersection a run whose start column is the start column of run $i+1$ and whose end column is the greater of the end columns for runs i and $i+1$. This algorithm relies on the assumption that the intersection of any three runs in the sorted run list is zero. This assumption follows from the fact that we are taking the intersection of exactly two image objects and from our assumption that the runs specified for a single image object are disjoint.

In the worst case, the number of rows occupied by both image objects is equal to $\min(n_{rl}, n'_{rl})$. The computational complexity is simply this number times the total number of runs in each sorted run list, that is, $O(\min(n_{rl}, n'_{rl}) \times (n_r + n'_r))$. This computational effort must be added to that of comparing and sorting the images.

Algorithm	Computational Complexity
Grid to RLE	N
Grid to Poly	N
RLE to grid	r
RLE to poly	r
Poly to grid	e
Poly to RLE	e

Notation: N = number of cells in grid
r = number of runs in RLE
e = number of edges in polygon

Table 4-2: Summary of Update Algorithm Complexity Results

4.2.3.3 Negation

If they are not already sorted, sort the runs in each run list in order of increasing start column. For each row that has no run list, NOT includes a run consisting of the entire row (start column = image-start-column, end column = image-end-column). Then for each run list, NOT includes the following runs, which we specify by the start and end columns. Run 1: image-start-column, run 1 start column -1 (assuming the latter is not less than the former). Run 2: (end-column run 1) $+1$, (start-column run 2) -1. Run i: (end-column run $i-1$) $+1$, (start-column run i) -1. Run r_r+1 (final run): (end-column run n_r) $+1$, (image-end-column).

If we assume that this algorithm is carried out within a bounding box of the image, and that the runs in each run list and the run lists themselves are sorted, then the computational complexity is on the order of the total number of runs in the image, that is, $O(n_{r_l} n_r)$.

4.2.4 Update Algorithms

The computational complexity of the update algorithms presented in Chapter 3 is shown in Table 4-2. In each case, it is the representation complexity of the input that determines the complexity of the algorithm. Notice also that all the algorithms are linear in the complexity of the input.

4.2.4.1 Grid to RLE

The conversion from grid to RLE is straightforward. The algorithm must examine each grid row, one cell at a time and take the appropriate action depending on the value of the cell and whether or not a run is currently being formed. Since every cell must be examined, the algorithm is $O(N)$ where N is the number of cells in the grid.

4.2.4.2 Grid to Polygon

Grid to polygon is more expensive than grid to RLE. The algorithm must first determine the connectivity of the grid cells, creating a connected components grid. This involves one pass over every cell in the grid with a $3X3$ mask. The cost of this operation is $O(N)$. The next stage involves tracing the boundaries of the regions. This step is $O(n_b)$ where n_b is the number of cells on the region boundaries. Since the number of boundary points is less than the total number of grid points, we have a total complexity of $O(N)$ for the combined operation.

4.2.4.3 RLE to Grid

The RLE to grid operation simply iterates over the runs in the RLE depositing 1's in the grid for each point in the run. The resulting complexity is $O(r)$, where r is the number of runs in the RLE.

4.2.4.4 RLE to Polygon

The algorithm for converting from RLE to polygon moves from run-list to run-list. keeping track of the next and previous run-lists. The individual runs in the list are compared against the runs in the next and previous run-lists to determine connectivity. Knowing the order of the run-lists allows this operation to be local and keeps the complexity linear in the number of runs $O(r)$.

4.2.4.5 Polygon to Grid

The polygon to grid algorithm intersects a fixed number of scan-lines with the polygon structure; the cost for this is se, where s is the number of scan-lines and e is the number of edges in the polygon. Since s is fixed, we have an analytic complexity of $O(e)$.

4.2.4.6 Polygon to RLE

The polygon to RLE algorithm is essentially the same as the algorithm for polygon to grid conversion. The complexity of the polygon to RLE operation is hence $O(e)$ as well.

4.3 Complexity of Data

4.3.1 Defining an Inherent Data Complexity Measure

While algorithmic computational complexity is a well-defined and in principle computable quantity, effective computational complexity can vary widely with the complexity of the data being operated on. Our goal is to develop a complexity measure for an underlying image data set so that we may correlate the effective algorithmic complexity with this image complexity measure.

We are not the first to consider the notion of complexity of images. In fact, the notion of complexity is implicit in pattern recognition and digital filtering. Most pattern recognition schemes begin with a 'cleanup' process involving the filling in of holes, smoothing of lines, connection of arcs, etc., each of which renders the image, in some intuitive sense, less complex. Grenander ([Gren 69],[Gren 70]) discusses in mathematical terms the properties that such simplifying operations should have and their relationship to the classification process. In digital filtering, constraints such as maximum entropy are imposed to determine a unique solution to the otherwise ill-posed inversion problem. There are several related complexity measures. Shannon entropy [Shan 49] is related to the information content of an image or of any 'message'. Grassberger [Gras 86] defines and discusses several complexity measures on dynamically generated (e.g., chaotic) patterns. Finally, Huberman *et. al.* ([Hube 86],[Cecc 87]) have defined a complexity measure for hierarchical structures which takes into account the diversity of substructures at each level of the hierarchy.

In the remainder of this chapter we discuss what we believe to be desirable properties of an image complexity measure as well as the factors which we believe should enter into the measure. Our discussion is consistent with the intuitive notion of simplicity/complexity used in pattern recognition as described above; it contains information theoretic elements as well.

4.3.1.1 Desired properties of the complexity measure

With respect to our goal of estimating computational complexity as a function of image complexity, an image complexity measure should have the following properties:

1. Scale invariance

   We would like to be able to compare the complexities of data sets of different sizes, that is, of different physical sizes or of different resolution. This requires an appropriately normalized measure. On a binary array, for example, complexity should be independent of changes in resolution (modulo the aliasing problem). Intuitively, we do not want complexity to depend on the size of an object, except perhaps as it relates to the size of other objects in the image. Rather, complexity should depend on the shape and topology of objects. (We give a precise definition of "object" in Section 4.3.1.3)

   In typical pattern recognition processes, after an image is 'cleaned up' as described above, the pattern is centered and brought to a standard scale so that it may be more easily classified. The point we would like to make here is that this scaling process does not change the identity of the pattern or object represented by the pattern. If complexity is an identifying property of a pattern, then, it should not be affected by simple linear scale changes in the pattern.

2. Reflection symmetry

   The complexity measure should be invariant under reversal of 0's and 1's (assuming a binary grey-scale). The premise here is that data is equally complex

for objects made up of 1's on a background of 0's as for the same objects made up of 0's on a background of 1's. For example, a field with scattered clumps of trees may have the same complexity as a forest with scattered clearings. The desire for reflection symmetry in the complexity measure is motivated by the belief that topological considerations should completely override absolute specifics such as whether a region is labeled by 0's or by 1's.

For a multivalued grey-scale, the notion of reflection symmetry generalizes straightforwardly to permutation symmetry. An image attribute is permutation symmetric if its value is unchanged under permutation of grey-scale value labeling.

4.3.1.2 Image attributes entering into the complexity measure

The complexity measure must in some general way reflect how difficult it will be to perform computations on the data. To achieve this, we would like to compress the complete set of image data into a smaller set of image attributes which will be sufficient (together with a measure of algorithmic complexity) for estimating the computational complexity of performing an algorithm on the input image data. Furthermore, we may characterize an image attribute according to whether it is on the pixel level, the 'object' or group-of-pixels level, or the group-of-objects level. An attribute is said to be at a certain level if it can be computed by considering items only at that level or at lower levels. Candidate image attributes include:

1. Number of objects (object level)

   The greater the number of objects, the greater the number of separate things we need to keep track of.

2. Image entropy (pixel level)

   Computations are relatively easily done on images with little information content. For example, calculations are trivial when carried out on images that are all white or all black. In some average sense it should be increasingly difficult to perform computations on arrays of increasing information content.

3. Object compactness (object level)

   The less compact, or 'stringier', an object is, the more difficult it is to compute with. We might alternatively or additionally have a term in the complexity measure which is a function of the amount of information required to define an object. (In practice, however, this might be difficult to compute because it would entail determining the symmetries of each object.)

4. Clustering of objects (group-of-objects level)

   Effective computational complexity will be different in the case that objects are clumped in the corner of an image than when they are distributed uniformly throughout the image.

Of course, there are many other image attributes that could be considered in the complexity analysis. For example, at the object level, Levine [Levi 85] lists these:

- Number of vertices.
- Variance in the lengths of sides of the figure, or ratio of maximum to minimum side length.
- Boundary curvature.

Clearly, these object or shape complexity characteristics are not mutually independent, nor are they completely redundant. For simplicity,the complexity analysis uses object compactness as the only the shape descriptor. The analysis also includes the image descriptors number of objects and image entropy.

4.3.1.3 Definitions for complexity attributes

The previous section motivated the choice of image attributes on which the image complexity measure depends. No one of these attributes has a unique definition. In this section we discuss our specific choices for the definitions of these terms.

1. **Number of objects**
   An *object* is a region of the image composed of 0's (or 1's) which is bordered entirely by a region of 1's (or 0's) or by the edge of the image. In the more general case of a multivalued grey-scale, an object is simply any connected component. In this way every region of unique grey-scale value (or, for a discrete image data set, every cell in the image) belongs to an object.

   The motivation for this is that without *a priori* knowledge of which regions or objects are of interest, every connected component is of potential interest. As noted in the discussion of reflection/permutation symmetry in Section 4.3.1.1, it does not make sense to assign arbitrarily as objects those connected components with specific grey-scale values.

2. **Image entropy**
   Image *entropy* is defined simply to be Shannon entropy. That is,

$$\text{entropy} = \sum_i p_i \log_2 p_i,$$

   where p_i is the fraction of the image with grey-scale level value i. This definition is clearly symmetric under permutation of grey-scale level values.

   For a binary image, i is either 0 or 1 and $p_0 = 1 - p_1$. Entropy is then equal to 0 when $p_0 = 0$ or $p_1 = 0$, and is a maximum at the midpoint $p_0 = p_1 = 1/2$. As discussed in Section 4.3.1.1, this is the desired behavior.

3. **Object compactness**
   The *compactness* of an image object to be

$$\text{compactness} = \text{perimeter}/\sqrt{\text{area}},$$

where *perimeter* and *area* are, for a binary array, measured in units of grid spacing (i.e., the length of a grid cell edge) and number of grid cells respectively. The reason for dividing *perimeter* by $\sqrt{area}$ rather than by, say, *area* is invariance under linear scale transformations. For example, a square that is n on a side has the same value of (perimeter/$\sqrt{\text{area}}$) as one of side m for any n and m.

Note that for digitized images this measure of object compactness is commonly used with *perimeter* taken to be the number of cells on the border of the object, rather than the length of this border. The difference between these two measures of perimeter arises at 'corner' pixels. If the value of *perimeter* is the number of cells on the object border, then the compactness measure given above is not invariant under a linear change of scale.

To see that the compactness measure (with perimeter equal to the length of the object border) is invariant under a linear change of scale, note that any change of scale is equivalent to simply changing the yardstick with which measurements are made. For example, say perimeter is measured in some linear units called unit_1, so that perimeter = p unit_1. Rescaling the image by using a different unit of length unit_2 where $\text{unit}_2 = b$ unit_1 for some positive scale factor b. In these new units, perimeter = p/b unit_2. Similarly if area = a unit_1, then after the change of scale, area = a/b^2 unit_2. However, the combined quantity (perimeter/$\sqrt{\text{area}}$) is *dimensionless* and therefore does not depend on the scale factor b. Compactness as defined above is therefore a scale invariant quantity.

Given this definition of compactness, we must still define *perimeter* and *area* of connected components. Whereas the definition of area is straightforward, that of perimeter is not, particularly in the case of objects with holes (i.e., with multiple boundaries) and of objects which border in part on the edge of the image. The following convention will clarify this. The perimeter of an object is defined to be the summed lengths of all boundaries of the object. If some of an object's boundaries begin and end on the edge of the image, then the perimeter includes the length of that portion of the edge which connects the boundary ends such that the object is enclosed (see Figure 4-2).

Finally, note that in practice, the quantity of interest is

$$\langle \text{perimeter}/\sqrt{\text{area}} \rangle$$

where $\langle \ldots \rangle$ represents an average over all objects in the image. In the definition of perimeter above, each boundary of the image is used exactly twice in this averaging process. One may define perimeter in such a way that boundaries are used only once in the averaging process. This was deemed less preferable, however, since it entails having qualitatively different definitions of perimeter as a function of boundary topology.

4. **Clustering of objects**

One measure of object clustering is simply the fluctuation or standard deviation

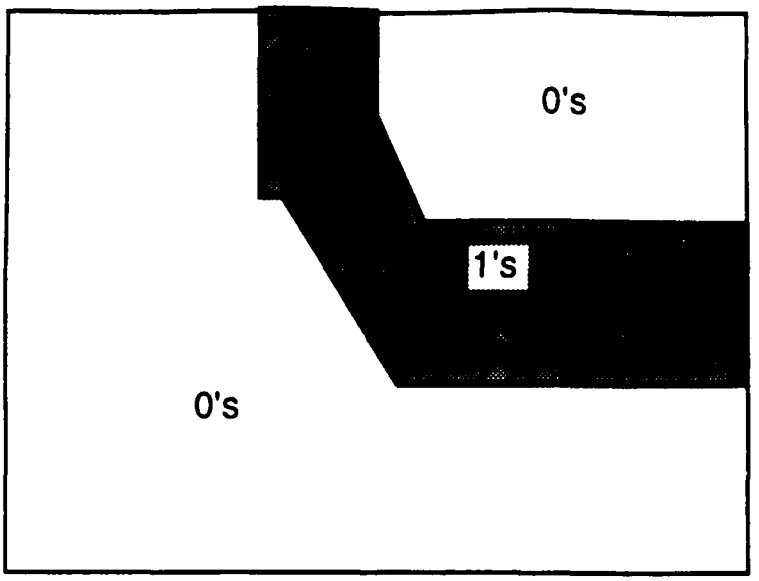


Figure 4-2: Boundary of Object

in position of object centroids. We do not discuss this further since we do not make use of a clustering of objects measure in what follows.

4.3.1.4 Image complexity measure

The previous sections defined a space of image attributes. This section we uses these attributes to define a single value representative of the image complexity. It combines the number of objects (N), the image entropy (E) and the mean object compactness (c) into a single complexity number (C) calculated as follows:

$$C = NE\frac{\sum_{i=1}^{N} c_i}{N}.$$

While this complexity measure may or may not correspond to human intuition about the complexity of images (see Section 4.3), its validity as a complexity measure must be evaluated empirically. To be a good measure of complexity it must:

- be a good predictor of algorithm performance,
- be computable from any representation,
- be independent of representation,
- be quickly computed.

Algorithm performance is usually specified in terms of representation complexity, hence if our complexity measure is closely correlated with various representation complexities it will likely be a good predictor of algorithm performance.

Name	*Dataset*	*No. Objects*	*Complexity*	*Theme*
cnpy0		55	high	all canopy
cnpy1		12	low	Canopy: 0-25
cnpy2		22	med	Canopy: 25-50
cnpy3	3A	15	8.95	Canopy: 50-75
cnpy4	6A	40	170.06	Canopy: 75-100
obst1		21	low	all obstacles
trav1		8	low	Cross Cntry Movement: 1
trav2	7B	67	359.59	Cross Cntry Movement: 2
trav3	7A	78	378.86	Cross Cntry Movement: 3
trav4	3B	20	9.85	Cross Cntry Movement: 4
trav5	1A	5	0.29	Cross Cntry Movement: 5
trav6	6B	40	177.04	Cross Cntry Movement: 6
trav7	4A	23	20.39	Cross Cntry Movement: 7
typ1		35	high	Agriculture, cropland
typ2	4B	23	36.97	Grassland, pasture, meadow
typ4	5B	41	97.83	Coniferous forest
typ5		29	med	Deciduous forest
typ6	5A	39	70.61	Mixed forest
typ7	2B	8	1.48	Forest clearings, cutover areas
typ17	1B	3	0.05	Bare ground, sand dunes
typ29	2A	11	2.96	Villages
typ30		13	low	Towns

Table 4-3: CATTS Data from which selections were made.

4.3.2 Data Selection

To be effective, the data selected to be used in the performance analyses of this project has to conform to several constraints. First, it must to be real data. Since the results of this research are to be used in terrain analysis systems that operate over real terrain data, it was felt that "theoretical" or otherwise "tuned" data would not be truly representative. Consequently, a variety of feature datasets of CATTS data from the Fulda Gap region of Germany were used to make the selections of data to be used in the studies (Table 4-3).

Secondly, a range of data complexities was chosen. This is based on the hypothesis that the efficiency of algorithms and data structures "tuned" to different data structures will vary depending on data complexity. That is, some algorithms will work better on "low" complexity data than "high" complexity data. Consequently, the complexity of the application data may determine the choice of data structure. Such determinations are the goal of this study.

Name	*Dataset*	*Objects*	*Entropy*	*Compactness*	*Complexity*
trav5	1A	5	0.0102	5.732	0.292
typ17	1B	3	0.0036	4.663	0.050
typ29	2A	11	0.0407	6.613	2.959
typ7	2B	8	0.0350	5.297	1.482
cnpy3	3A	15	0.0968	6.160	8.951
trav4	3B	20	0.0750	6.568	9.846
trav7	4A	23	0.1272	6.965	20.387
typ2	4B	23	0.1883	8.534	36.966
typ6	5A	39	0.2776	6.524	70.614
typ4	5B	41	0.3543	6.734	97.827
cnpy4	6A	40	0.5609	7.579	170.056
trav6	6B	40	0.5907	7.493	177.039
trav3	7A	78	0.6350	7.649	378.875
trav2	7B	67	0.6389	8.401	359.591

Table 4-4: The selected datasets.

4.3.3 Inherent Data Complexity of Datasets

The data complexity measure described in 4.3.1 was used to select 14 datasets to make up the test suite. The data was originally available only in grid form and was converted as necessary to the other data representations being used.

The selection took place in two steps. First, the original candidates (Table 4-3) were plotted and ranked by visual inspection since the data complexity measure generally corresponds to human intuition. Then the data complexity measure was computed for "likely" datasets and examined to make sure that

- The paired datasets had similar complexity values
- The seven paired datasets covered a wide range of complexity values
- The spacing between complexity values for the different datasets was fairly constant

The results are shown in both tabular (Table 4-4) and graphic form in 4-3.

Pairs of datasets are required for some algorithms such as the intersection and union operations. The datasets were selected from the same geographic location within a single terrain database and often from the same thematic field. Since a given theme partitions the world into disjoint regions, many of the chosen datasets fit together like pieces of a jigsaw puzzle. For boolean set operations this creates uninteresting and artificial results. To provide more interesting test data three new datasets were created from three of the original datasets by either shifting or transposing the real image's coordinates. For example, Figure 4-3 shows dataset 6A. Switching the X and Y axis we creates a new image (Figure 4-4) with the same complexity as the original image but with a different description.

Figure 4-3: Dataset 6A

Figure 4-4: Dataset 6I: 6A with X and Y axis exchanged.

	DATA	INHERENT COMPLEX	GRID COMPLEX	RLE COMPLEX	POLY COMPLEX
1	1A,1S	.584	700.000	139.932	92.000
2	2A,3I	11.910	6945.000	562.884	492.000
3	4A,3I	29.338	12471.000	1006.938	887.000
4	5A,3I	79.565	26005.000	1556.808	1317.000
5	6A,3I	179.007	70361.000	2707.064	2209.000
6	4A,6I	190.443	72504.000	3004.130	2490.000
7	5A,6I	240.670	86038.000	3554.000	2920.000
8	7A,3I	387.826	91989.000	3894.904	2994.000
9	7A,6I	548.931	152022.000	5892.096	4597.000

Table 4-5: Paired Datasets and Combined Complexity

Similarly, dataset 3I was created by transposing the X and Y axis of dataset 3A (see appendix, Figure 7-9). An additional test dataset was created from dataset 1A which was shifted by three pixel locations in the negative X and positive Y directions to produce dataset 1S (see appendix, Figure 7-8).

Then, the datasets were paired to produce a range of combined complexities (Table 4-5). Pairs were created from datasets of high and low complexity, low and low complexity, high and high complexity, etc.

4.3.4 Representation Complexity of Datasets

The analytic complexity of the chosen algorithms (see Section 4.2) determines properties of a representation affect processing time. The key properties of the representations are:

- **Grid**: the number of grid cells N and the number of cells corresponding to regions n_o,
- **RLE**: the number of runs n_r and the number of run lists n_{rl},
- **Polygon**: the number of edges n_e.

To compute the representation complexity of each test dataset (1A - 7A) the dataset is instantiated in each of the representations (Grid, RLE, and Polygon). The appropriate representation properties are measured, and the results tabulated (see Tables 4-6 -- 4-8).

	COMPLEXITY MEASURE	1A	2A	3A	4A	5A	6A	7A
1	NO. OF POINTS	350	1781	5164	7307	20841	65197	86825
2	N x N	262144	262144	262144	262144	262144	262144	262144

Table 4-6: Grid Representation Complexity of Datasets

	COMPLEXITY MEASURE	1A	2A	3A	4A	5A	6A	7A
1	RUN LISTS	69.000	156.000	276.000	338.000	438.000	512.000	512.000
2	AVERAGE NO. RUNS PER LIST	1.014	1.333	1.286	1.929	2.744	4.594	6.914
3	TOTAL RUNS (ROW1 * ROW2)	69.996	207.948	354.936	652.002	1201.872	2352.128	3539.968

Table 4-7: RLE Representation Complexity of Datasets

	COMPLEXITY MEASURE	1A	2A	3A	4A	5A	6A	7A
1	NO. OF POLYGON EDGES	46	189	303	584	1014	1906	2691

Table 4-8: Polygon Representation Complexity of Datasets

4.4 Analysis of Data Complexity

Given knowledge of the analytic complexity of an algorithm and the representation complexity of the inputs, one can predict the expected relative performance of the algorithm. The question is whether or not inherent data complexity likewise provides a good predictor of algorithm performance. If there exists a statistically significant relationship between inherent data complexity and representation complexity then data complexity is likely to be a valuable predictor.

Figures 4-5 through 4-7 graphically illustrate the relationship between representation and inherent data complexity. The graphs show the relationship between representation complexity and data complexity, as well as the components of the data complexity measure (entropy, compactness, and number of objects). As can be seen in these graphs, there is a roughly linear relationship between representation and data complexity. This means that as representation complexity increases we can expect a linear increase in the inherent data complexity.

To test whether or not inherent data complexity is a good predictor in practice, we must generate performance models for the given algorithms using both representation and inherent data complexity. If both measures give roughly equivalent performance models which have similar accuracies, then we will have empirical verification that data complexity can be used to predict performance.

4.5 Performance Models of Algorithms

4.5.1 Test Environment

The goal of this portion of the Phase II research is to develop performance models of the selected algorithms. The methodology for generating performance models is as follows:

- set up a baseline test environment
- run the algorithms against the selected datasets,
- calculate the running times for the algorithms,
- tabulate the results of running time verses representation complexity and inherent data complexity,
- fit a curve to the data using expectations of algorithm performance,
- compare curves generated using the two complexity measures.

4.5.1.1 Running and Timing the Algorithms

The algorithms used in this study were developed on a SUN 3/110 running Version 3.5 of the UNIX operating system, with 12 megabytes of available memory. The specific programming environments were Lucid LISP Version 3.0 and the Common

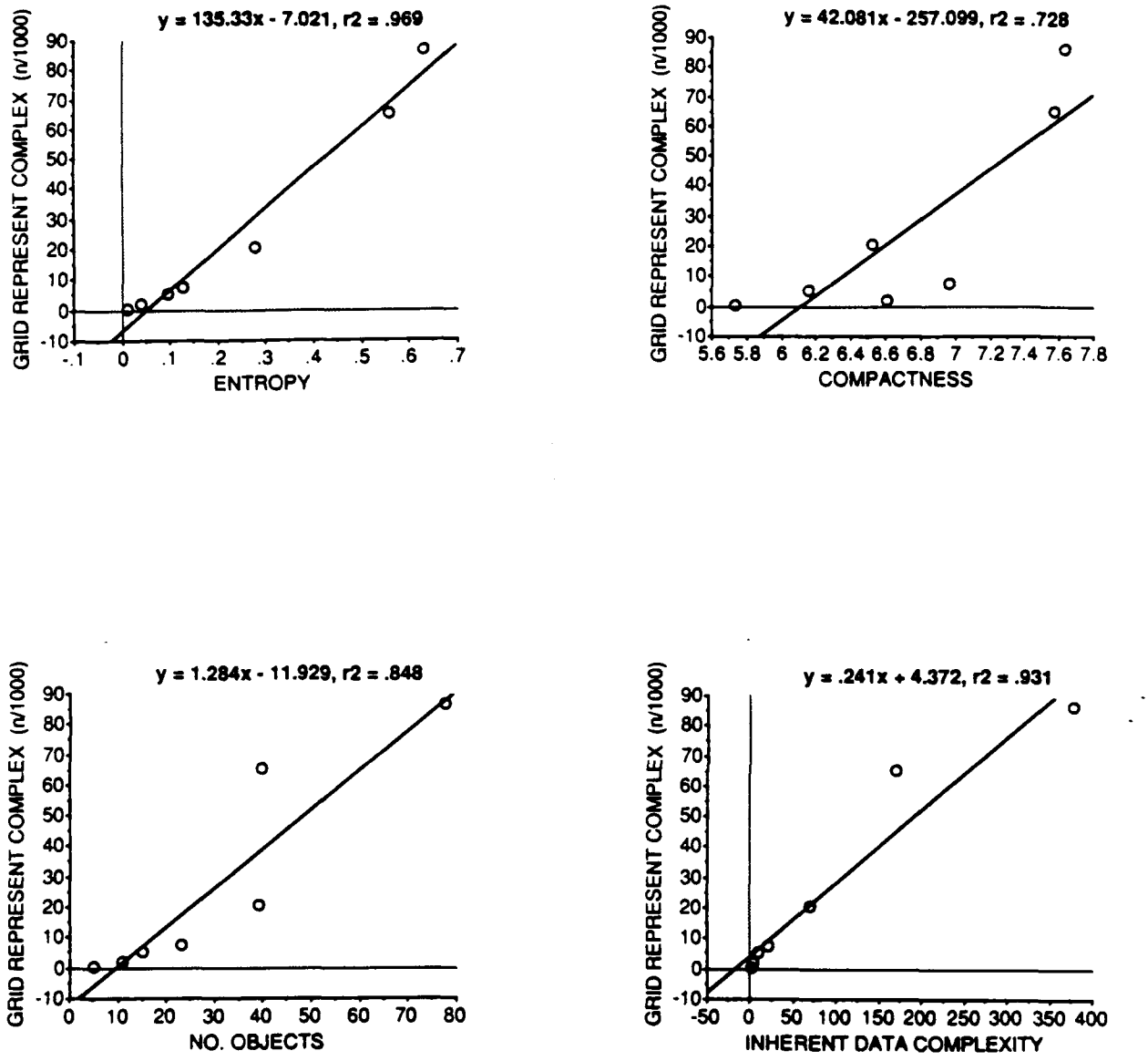
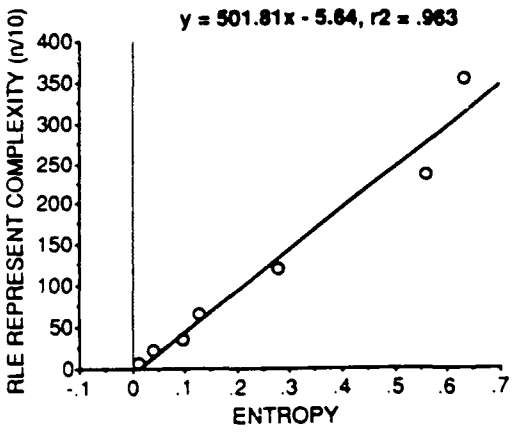
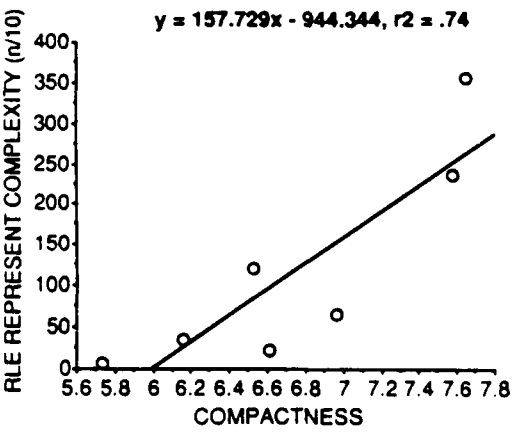
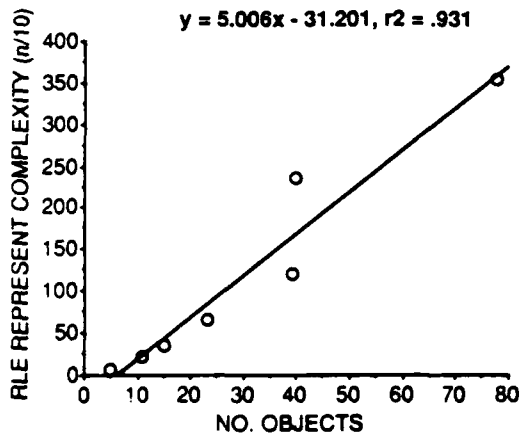


Figure 4-5: Grid Representation Complexity *vs* Inherent Data Complexity







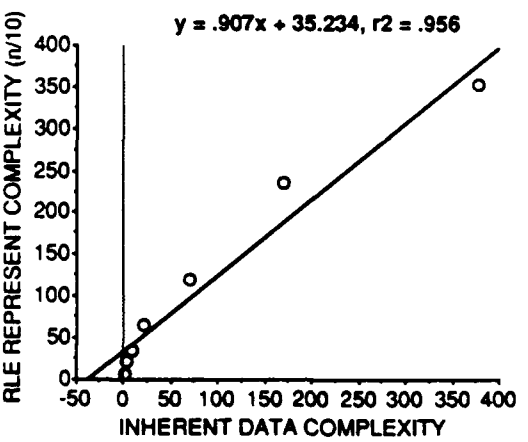
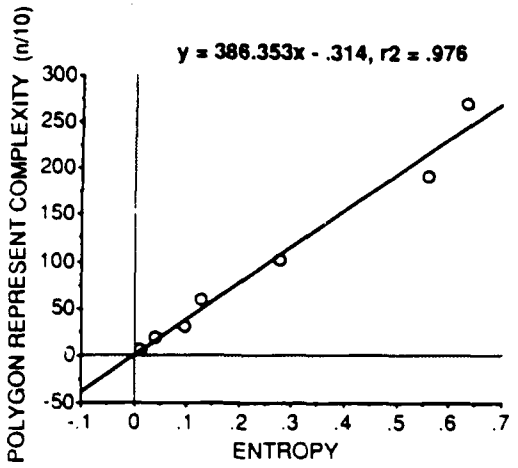
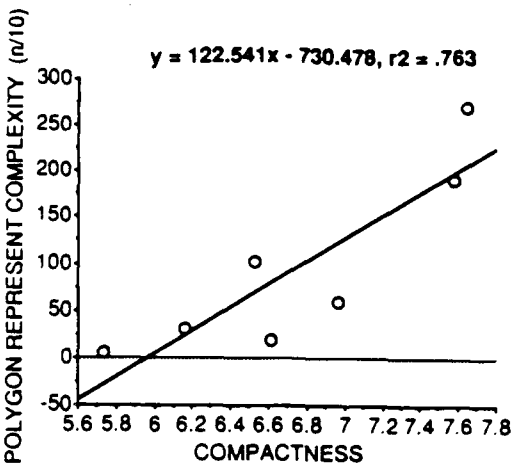
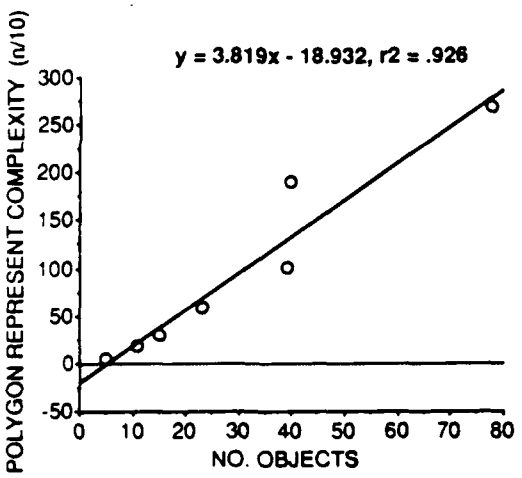


Figure 4-6: RLE Representation Complexity *vs* Inherent Data Complexity







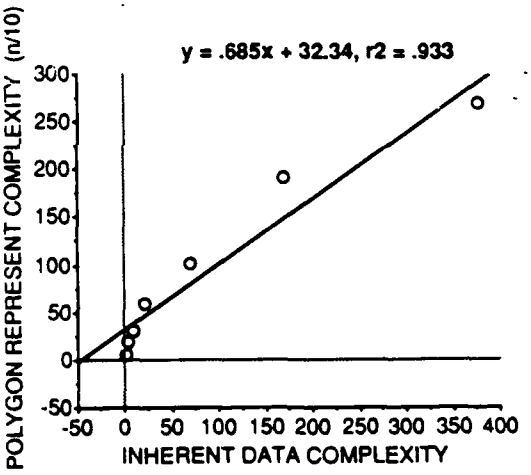


Figure 4-7: Polygon Representation Complexity *vs* Inherent Data Complexity

Lisp Object System (CLOS) [Bobr 88]. The basic timing tool used in this performance study is a macro supplied in Lucid LISP.

The Lucid macro "time" was used to measure the CPU time of each execution of an algorithm. The timing function provides a rough break-down of where processing time was spent. The following is an example of the timing function applied to the grid intersection operation:

```
Grid intersect.
;;; GC: 542672 words [2170688 bytes] of dynamic storage in use.
;;; 1161262 words [4645048 bytes] of free storage available
;;; before a GC.
;;; 2865196 words [11460784 bytes] of free storage available
;;; if GC is disabled.
Elapsed Real Time = 94.08 seconds
Total Run Time    = 88.26 seconds
User Run Time     = 85.86 seconds
System Run Time   = 2.40 seconds
Dynamic Bytes Consed = 32776
There were 14 calls to GC
```

Notice that before the operation is run, the "Garbage Collector" is called to reclaim any free memory. The function is then run and the computed timings are displayed. The "User Run Time" value provides the most accurate measure of the processing time of the algorithm. Unfortunately, when running in a Lisp environment, it is not possible to completely isolate the running of the user process from that of system processes like *garbage collection.* While we have access to how often garbage collection was performed, we do not have a measure for the total time involved.

To alleviate some of the uncertainty involved in timing Lisp functions, algorithms were timed on a number of separate runs. The minimum run-time for each algorithm was selected, as this represents the closest to optimal performance of the algorithm. The algorithms were run up to five times each (Figure 4-9), and the test environment was restarted often to provide an uncorrupted LISP environment to insure the most accurate timings.

Additionally the order in which tests were run was altered periodically to ensure that this ordering had no impact on the timings. The times from one run of an algorithm to another differed very little. Occasionally extra time would be spent in garbage collection or system functions. By taking the minimum time, anomalies like these were removed from consideration.

Note that, for the purposes of this study, the absolute processing times of the algorithms are not especially significant. Instead, the important information is the relation of the run time of one algorithm to the run time of another algorithm. Given a more powerful computer, each of the algorithms would run much faster, but proportionately, their run time would be the same. Thus when analyzing the expected performance of an algorithm, it is important to use the predicted time only as a comparative figure and not an expected running time. This highlights the significance of having a baseline environment in which to perform these tests. Without a

	DATA SET	TRIAL 1	TRIAL 2	TRIAL 3	TRIAL 4	TRIAL 5	MIN TIME	MEAN TIME
1	1A	7.58	7.58	7.72	7.70	7.58	7.58	7.632
2	2A	32.28	32.36	32.36	32.08	32.06	32.06	32.228
3	3A	45.96	45.94	46.00	45.94	46.36	45.94	46.040
4	4A	90.80	91.74	90.82	90.78	91.58	90.78	91.144
5	5A	169.78	164.76	164.70	166.12	164.72	164.70	166.016
6	6A	340.84	340.62	339.08	381.58	376.84	339.08	355.792
7	7A	476.58	477.44	476.34	476.28	472.36	472.36	475.800

Table 4-9: Example Trials for RLE to Polygon Update Algorithm

common environment for each algorithm it is impossible to accurately compare their performance.

4.5.1.2 Computing a Performance Model (Regression)

The tests in this performance study involve two measurements; the processing time of the algorithm, and the complexity of the input. These measurements provide two variables which form a *bivariate population.* Given such a population, two questions arise:

- is there a relationship between the the variables?
- if they are related, what equation best expresses the relationship in some sense?

Plotting the test points on a graph with processing time on the Y axis and complexity on the X axis produces a *scatter diagram.* The goal is to find an equation which will minimize the square of the error between the observed values and the values predicted by the equation. This is known as the *method of least squares*, and the equation generated is known as the *regression of Y on X.*

Given the best equation expressing the relationship between the variables, the next step is to determine how useful it is for estimation. This satisfies the project goal; to be able to accurately estimate the processing time of an algorithm from the complexity of the input. This predictive equation yields a *performance model for the algorithm.* The predictive value of the equation is expressed by r, the *correlation coefficient.* The square of this (r^2), otherwise known as the *coefficient of determination*, because it has the useful property of always being between 0 and 1, where 1 indicates a perfect correlation and 0 indicates no predictive correlation at all.

In this study, the value of r^2 for each regression equation is a measure of the predictive value of the performance model. It is also of interest to measure the

	DATA	GRID->RLE	GRID->POLY	RLE->GRID	RLE->POLY	POLY->GRID	POLY->RLE
1	1A	21.720	29.360	3.240	7.580	11.300	7.960
2	2A	21.820	53.900	3.960	32.060	32.720	28.780
3	3A	21.880	67.720	5.600	45.940	52.460	47.080
4	4A	21.940	112.660	6.620	90.780	96.000	89.760
5	5A	21.980	186.820	13.480	164.700	172.120	160.080
6	6A	22.040	359.860	34.780	339.080	342.400	312.920
7	7A	22.160	498.240	45.320	472.360	484.640	446.660

Table 4-10: Update (Conversion) Algorithm Processing Times

difference between the values of r^2 achieved by performing the regression with either the representation complexity or the inherent data complexity on the X axis. If the difference in r^2 is small then both complexity measures will have similar predictive value. The expectation is that representation complexity will be a better predictor of performance that inherent data complexity as it is directly tied to the analytic complexity of the algorithm.

To actually compute the regression equations, a software package called "STATVIEW II" was used. This analytical and graphical program from ABACUS Systems, Inc. runs on the Apple MAC II. A table was initialized for each algorithm, in which the various time measurements for each dataset or dataset pair were entered, as well as the corresponding inherent data and representation complexities.

The minimum processing time for each dataset input into a specific algorithm was then selected as the y-coordinate, and each complexity measure as the x-coordinate of separate graphs. When computing the regression equations, the already known analytic complexity of the algorithms provides a starting point for the analysis. The analytic complexity bounds the algorithm's behavior - whether it is linear or polynomial for example.

In the tests that follow, the graphs of time versus both complexity measures are presented side by side. The r^2 values are presented, as is the difference Δr^2 between the r^2 for the performance model generated using representation complexity and the the one generated using inherent data complexity.

4.5.2 Update Algorithm Performance

The test results for the matrix of update algorithms applied to the test data are summarized in Table 4-10. In the following sections these results will be graphed to show the relationship between processing time and the two data complexity measures; representation complexity, and inherent data complexity. The curves generated by performing regression analysis on the data will be displayed, and the error in how well these curves match the data will be analyzed.

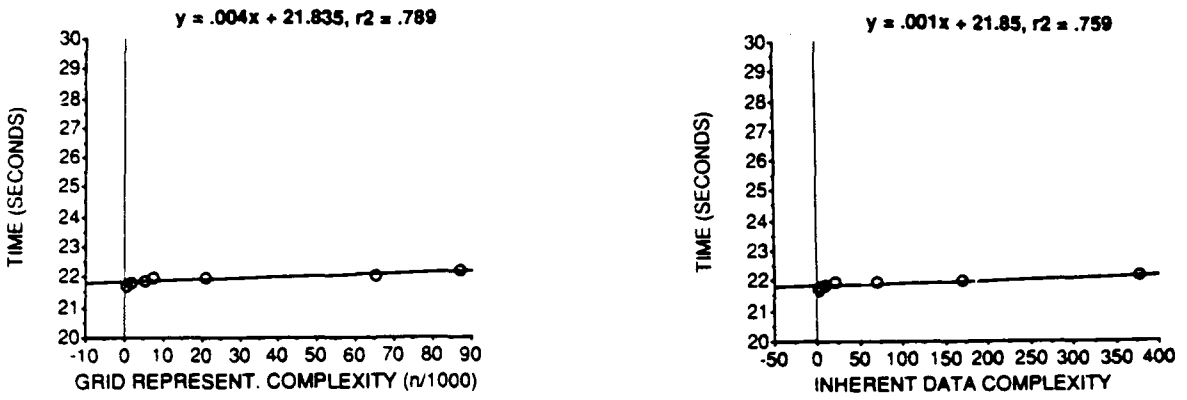


Figure 4-8: Grid to RLE Processing Time *vs* Complexity

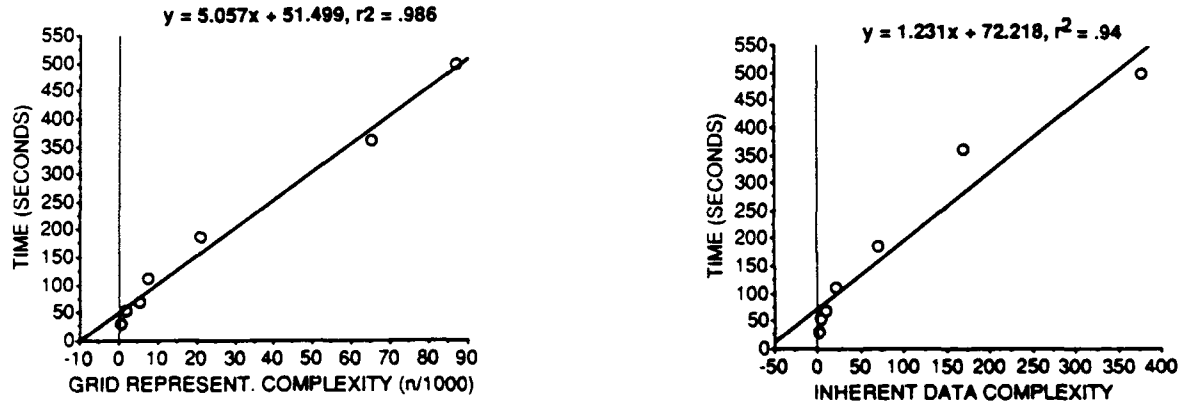


Figure 4-9: Grid to Polygon Processing Time *vs* Complexity

4.5.2.1 Grid to RLE

Figure 4-8 shows two graphs of processing time (in seconds) against the two different complexity measures; representation complexity on the left and inherent data complexity on the right. The correlation values and the difference between them are:

- correlation using representation complexity: $r^2 = 0.789$,
- correlation using inherent data complexity: $r^2 = 0.759$,
- difference of correlations $\Delta r^2 = 0.030$.

4.5.2.2 Grid to Polygon

Figure 4-9 shows two graphs of processing time (in seconds) against the two different complexity measures; representation complexity on the left and inherent

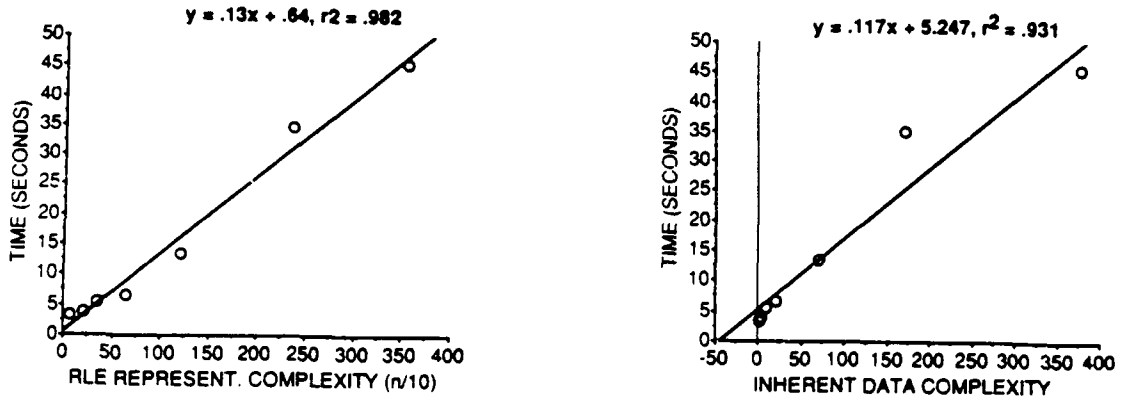


Figure 4-10: RLE to Grid Processing Time *vs* Complexity

data complexity on the right. The correlation values and the difference between them are:

- correlation using representation complexity: $r^2 = 0.986$,
- correlation using inherent data complexity: $r^2 = 0.940$,
- difference of correlations $\Delta r^2 = 0.046$.

4.5.2.3 RLE to Grid

Figure 4-10 shows two graphs of processing time (in seconds) against the two different complexity measures; representation complexity on the left and inherent data complexity on the right. The correlation values and the difference between them are:

- correlation using representation complexity: $r^2 = 0.982$,
- correlation using inherent data complexity: $r^2 = 0.931$,
- difference of correlations $\Delta r^2 = 0.051$.

4.5.2.4 RLE to Polygon

Figure 4-11 shows two graphs of processing time (in seconds) against the two different complexity measures; representation complexity on the left and inherent data complexity on the right. The correlation values and the difference between them are:

- correlation using representation complexity: $r^2 = 0.998$,
- correlation using inherent data complexity: $r^2 = 0.937$,
- difference of correlations $\Delta r^2 = 0.061$.

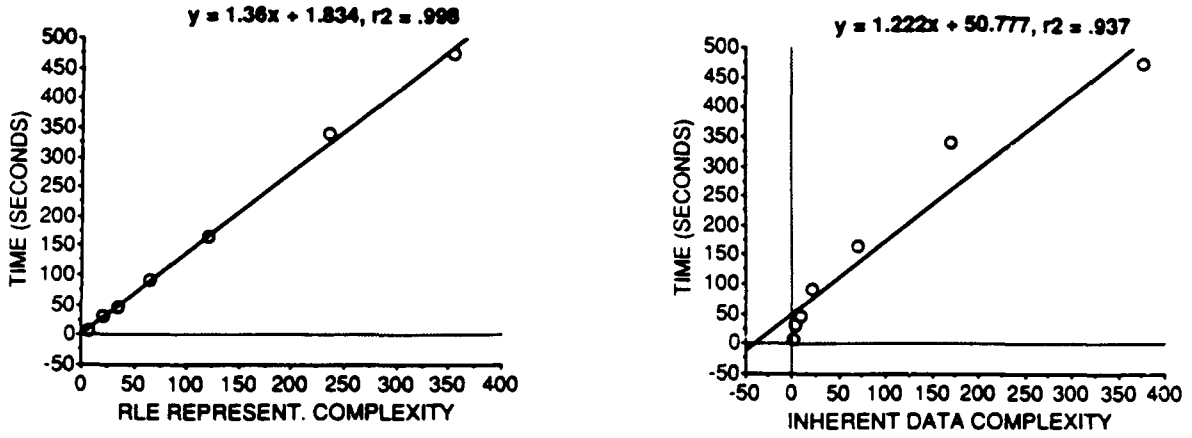


Figure 4-11: RLE to Polygon Processing Time *vs* Complexity

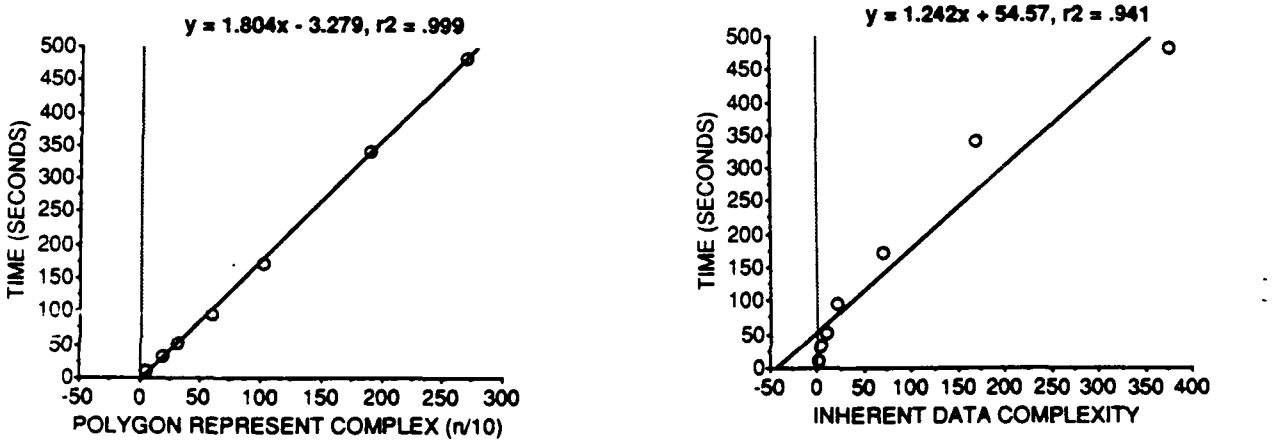


Figure 4-12: Polygon to Grid Processing Time *vs* Complexity

4.5.2.5 Polygon to Grid

Figure 4-12 shows two graphs of processing time (in seconds) against the two different complexity measures; representation complexity on the left and inherent data com$_{\text{i}}$ ·xity on the right. The correlation values and the difference between them are:

- correlation using representation complexity: $r^2 = 0.999$,
- correlation using inherent data complexity: $r^2 = 0.941$,
- difference of correlations $\Delta r^2 = 0.058$.

4.5.2.6 Polygon to RLE

Figure 4-13 shows two graphs of processing time (in seconds) against the two different complexity measures; representation complexity on the left and inherent

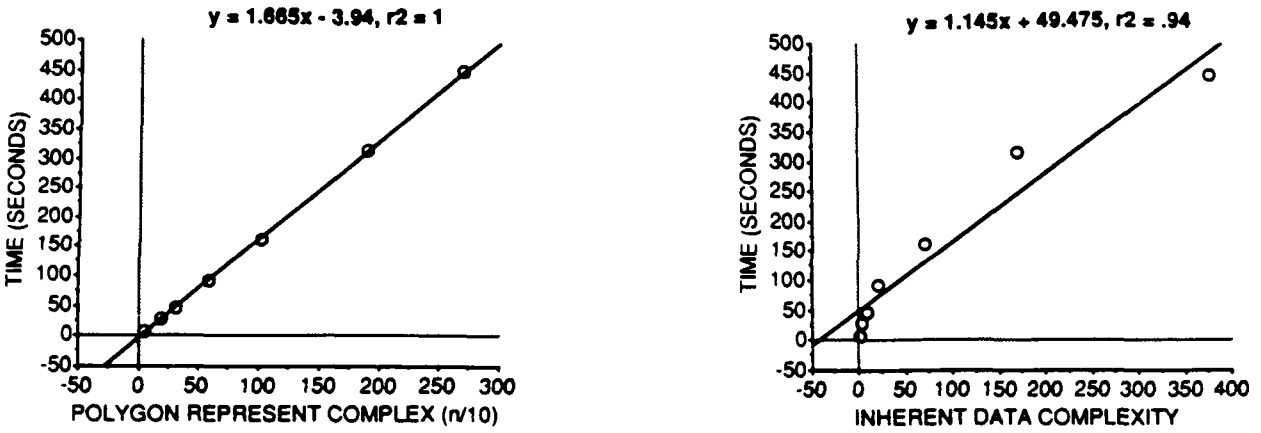


Figure 4-13: Polygon to RLE Processing Time *vs* Complexity

Method	*Performance Model*	*Correlation*
Grid To RLE	$y = .001x + 21.85$	$r^2 = .759$
Grid To Polygon	$y = 1.231x + 72.218$	$r^2 = .94$
RLE To Grid	$y = .117x + 5.247$	$r^2 = .931$
RLE To Polygon	$y = 1.222x + 50.777$	$r^2 = .937$
Polygon To Grid	$y = 1.242x + 54.57$	$r^2 = .941$
Polygon To RLE	$y = 1.145x + 49.475$	$r^2 = .94$

Table 4-11: Performance Models for Update Algorithms

data complexity on the right. The correlation values and the difference between them are:

- correlation using representation complexity: $r^2 = 1.000$,
- correlation using inherent data complexity: $r^2 = 0.940$,
- difference of correlations $\Delta r^2 = 0.060$.

4.5.2.7 Summary of Performance Models

The regression equations gnerated above are summarized in Table 4-11. The regression equations for the update algorithms have high correlation coefficients when using both representation and inherent data complexity. This means that performance models based on these equations will have a strong predictive value.

Also, the difference between the values of r^2 for the two complexity measures is never more then 0.06. This means that performance models generated using inherent data complexity are nearly as predictive as models generated using representation complexity.

	DATA	INHERENT C	GRID C	GRID T	RLE C	RLE T	POLY C	POLY T
1	1A,1S	.584	.700	85.520	13.993	.280	9.200	58.080
2	2A,3I	11.910	6.945	83.620	56.288	.440	49.200	69.600
3	4A,3I	29.338	12.471	85.820	100.694	.600	88.700	195.320
4	5A,3I	79.565	26.005	86.800	155.681	.720	131.700	215.140
5	6A,3I	179.007	70.361	90.020	270.706	.920	220.900	582.780
6	4A,6I	190.443	72.504	85.820	300.413	1.220	249.000	1665.400
7	5A,6I	240.670	86.038	86.800	355.400	1.760	292.000	3783.000
8	7A,3I	387.826	91.989	91.440	389.490	1.020	299.400	•
9	7A,6I	548.931	152.022	91.560	589.210	3.560	459.700	13225.580

Table 4-12: Intersection Processing Times for Grids, RLE's and Polygons

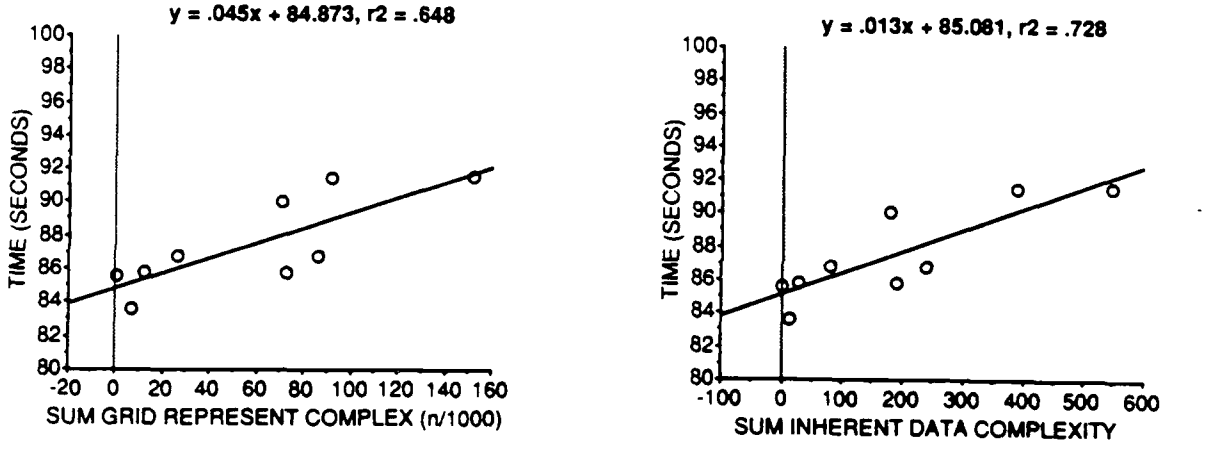


Figure 4-14: Grid Intersection Processing Time *vs* Complexity

4.5.3 Processing Algorithm Performance

4.5.3.1 Intersection

The test results for the intersection algorithms for the different representations applied to the test data are summarized in Table 4-12. The columns of the table contain the combined complexity of the input (for example **GRID C**) and the processing time for the algorithm in seconds (for example **RLE T**).

Figure 4-14 shows two graphs of grid intersection processing time (in seconds) against the two different complexity measures; representation complexity on the left and inherent data complexity on the right. The correlation values and the difference between them are:

- correlation using representation complexity: $r^2 = 0.648$,

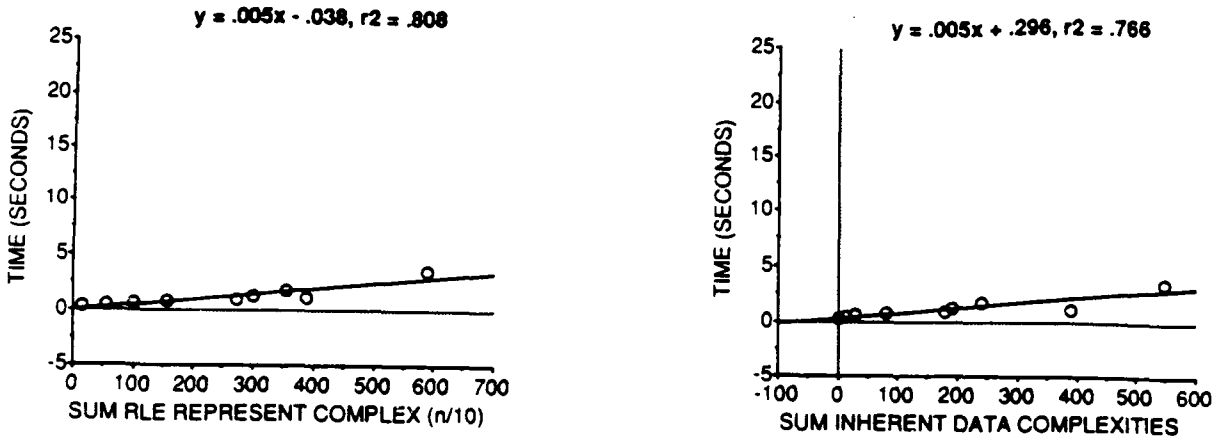


Figure 4-15: RLE Intersection Processing Time *vs* Complexity

- correlation using inherent data complexity: $r^2 = 0.728$,
- difference of correlations $\Delta r^2 = -0.08$.

The value for Δr^2 in this case is interesting because it indicates that inherent data complexity was a better predictor of processing time than was the representation complexity.

Figure 4-15 shows two graphs of RLE intersection processing time (in seconds) against the two different complexity measures; representation complexity on the left and inherent data complexity on the right. The correlation values and the difference between them are:

- correlation using representation complexity: $r^2 = 0.808$,
- correlation using inherent data complexity: $r^2 = 0.766$,
- difference of correlations $\Delta r^2 = 0.042$.

Figure 4-16 shows two graphs of polygon intersection processing time (in seconds) against the two different complexity measures; representation complexity on the left and inherent data complexity on the right. The correlation values and the difference between them are:

- correlation using representation complexity: $r^2 = 0.991$,
- correlation using inherent data complexity: $r^2 = 0.985$,
- difference of correlations $\Delta r^2 = 0.006$.

4.5.3.2 Union

The test results for the union algorithms for the different representations applied to the test data are summarized in Table 4-13.

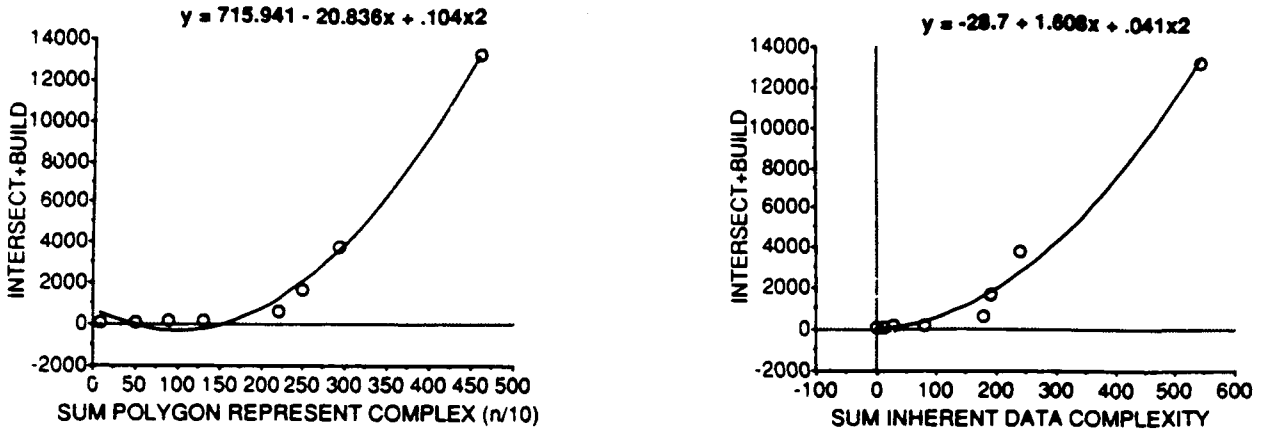


Figure 4-16: Polygon Intersection Processing Time *vs* Complexity

	DATA	INHERENT C	GRID C	GRID T	RLE C		POLY C	POLY T
1	1A,1S	.584	.700	103.880	13.993	.300	9.200	59.720
2	2A,3I	11.910	6.945	101.680	56.288	.600	49.200	99.420
3	4A,3I	29.338	12.471	101.880	100.694	.980	88.700	254.740
4	5A,3I	79.565	26.005	102.260	155.681	1.180	131.700	345.980
5	6A,3I	179.007	70.361	99.220	270.706	1.640	220.900	791.380
6	4A,6I	190.443	72.504	103.200	300.413	2.540	249.000	1974.280
7	5A,6I	240.670	86.038	102.280	355.400	3.620	292.000	4379.880
8	7A,3I	387.826	91.989	97.640	389.490	2.140	299.400	•
9	7A,6I	548.931	152.022	97.600	589.210	6.020	459.700	11256.940

Table 4-13: Union Processing Times for Grids, RLE's and Polygons

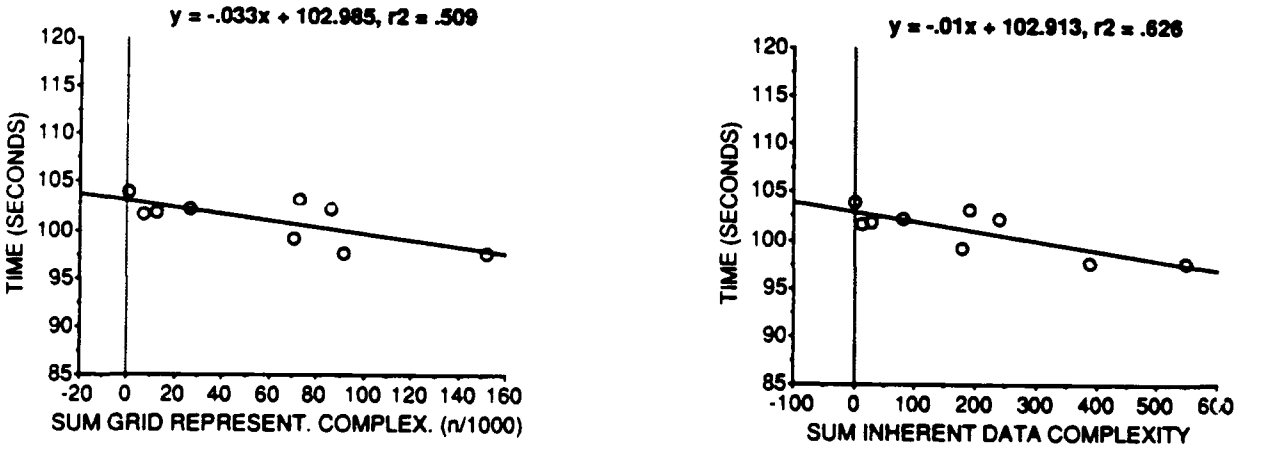


Figure 4-17: Grid Union Processing Time *vs* Complexity

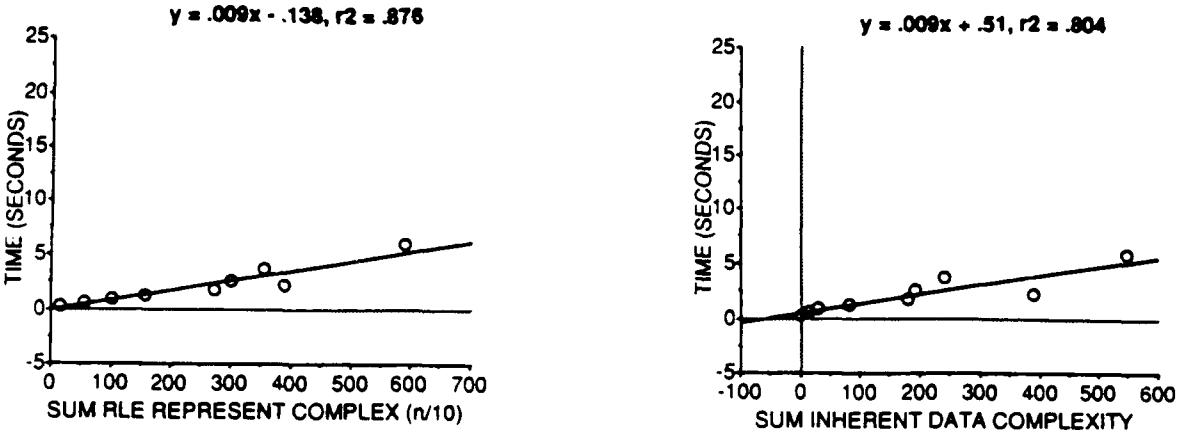


Figure 4-18: RLE Union Processing Time *vs* Complexity

Figure 4-17 shows two graphs of grid union processing time (in seconds) against the two different complexity measures; representation complexity on the left and inherent data complexity on the right. The correlation values and the difference between them are:

- correlation using representation complexity: $r^2 = 0.509$,
- correlation using inherent data complexity: $r^2 = 0.626$,
- difference of correlations $\Delta r^2 = -0.117$.

Once again inherent data complexity was a better predictor of algorithm performance. These results indicate that for grid union and intersection, some other property of the representation, in addition to the one measured, is affecting algorithm performance. This is likely due the use of a weak property of grids as the measure of representation complexity. We have used the number of grid cells in the region as the complexity measure, when, in fact, the time for grid union is roughly constant for grids of the same size.

Figure 4-18 shows two graphs of RLE union processing time (in seconds) against the two different complexity measures; representation complexity on the left and inherent data complexity on the right. The correlation values and the difference between them are:

- correlation using representation complexity: $r^2 = 0.876$,
- correlation using inherent data complexity: $r^2 = 0.804$,
- difference of correlations $\Delta r^2 = 0.072$.

Figure 4-19 shows two graphs of polygon union processing time (in seconds) against the two different complexity measures; representation complexity on the left and inherent data complexity on the right. The correlation values and the difference between them are:

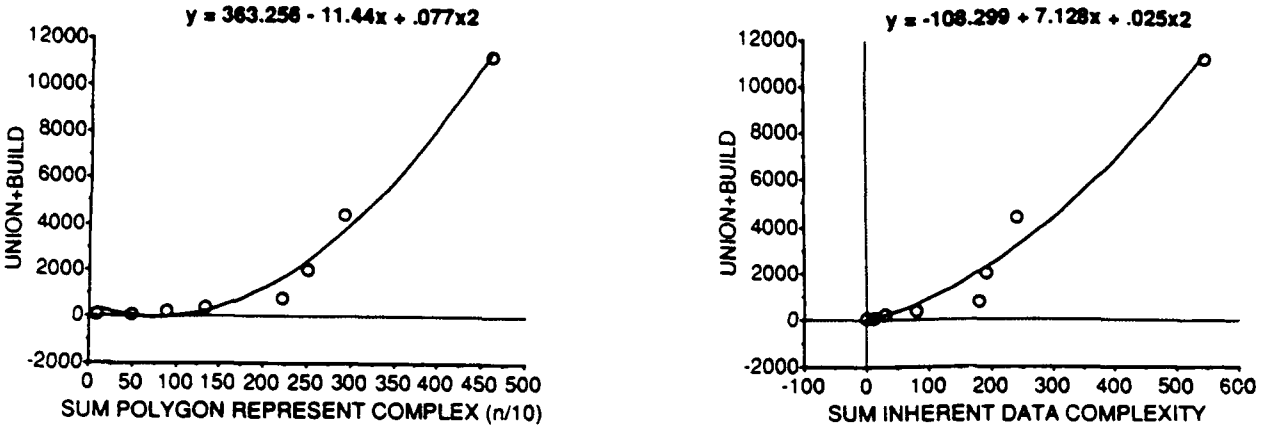


Figure 4-19: Polygon Union Processing Time *vs* Complexity

	DATA	INHERENT C	GRID C	GRID T	RLE C	RLE T	POLY C	POLY T
1	1A	.292	.350	87.600	6.997	.140	4.600	.200
2	2A	2.959	1.781	87.440	20.795	.220	18.900	.280
3	3A	8.951	5.164	87.320	35.494	.260	30.300	.320
4	4A	20.387	7.307	87.360	65.200	.380	58.400	.400
5	5A	70.614	20.841	87.480	120.187	.520	101.400	.520
6	6A	170.056	65.197	87.340	235.213	.860	190.600	.680
7	7A	378.875	86.825	87.240	353.997	1.180	269.100	1.140

Table 4-14: Negation Processing Times for Grids, RLE's and Polygons

- correlation using representation complexity: $r^2 = 0.985$,
- correlation using inherent data complexity: $r^2 = 0.968$,
- difference of correlations $\Delta r^2 = 0.017$.

4.5.3.3 Negation

The test results for the negation algorithms for the different representations applied to the test data are summarized in Table 4-14.

Figure 4-20 shows two graphs of grid negation processing time (in seconds) against the two different complexity measures; representation complexity on the left and inherent data complexity on the right. The correlation values and the difference between them are:

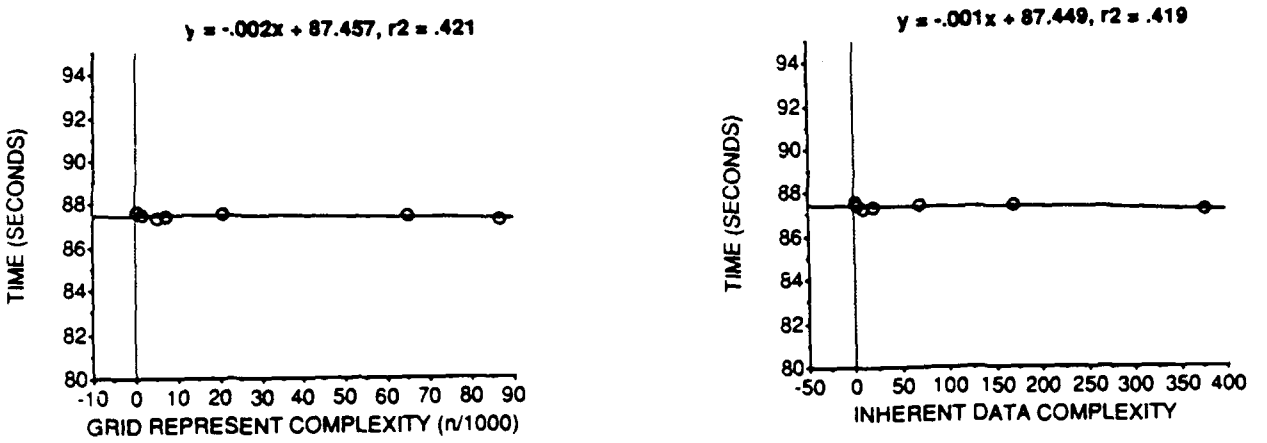


Figure 4-20: Grid Negation Processing Time *vs* Complexity

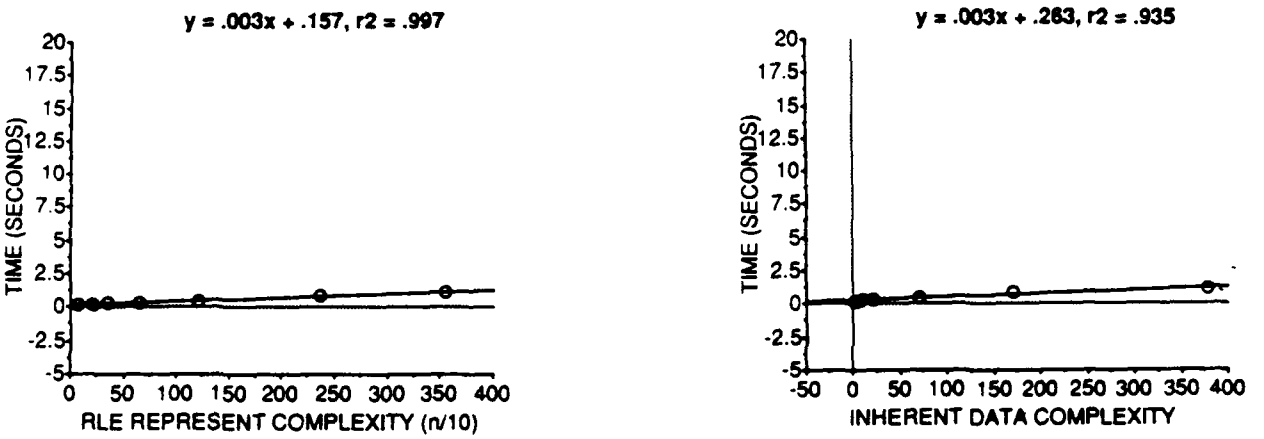


Figure 4-21: RLE Negation Processing Time *vs* Complexity

- correlation using representation complexity: $r^2 = 0.421$,
- correlation using inherent data complexity: $r^2 = 0.419$,
- difference of correlations $\Delta r^2 = 0.003$.

Figure 4-21 shows two graphs of rle negation processing time (in seconds) against the two different complexity measures; representation complexity on the left and inherent data complexity on the right. The correlation values and the difference between them are:

- correlation using representation complexity: $r^2 = 0.997$,
- correlation using inherent data complexity: $r^2 = 0.935$,
- difference of correlations $\Delta r^2 = 0.062$.

Figure 4-22 shows two graphs of polygon negation processing time (in seconds) against the two different complexity measures; representation complexity on the left

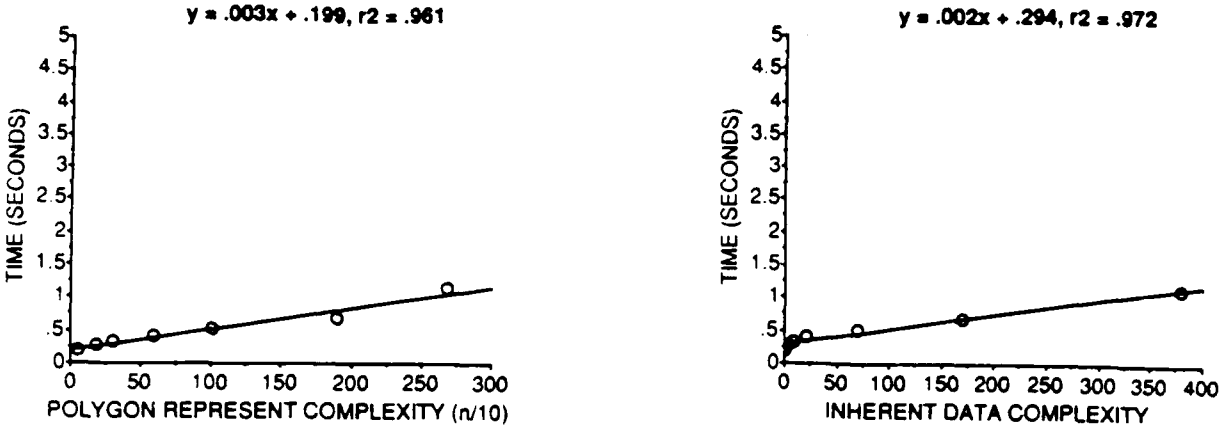


Figure 4-22: Polygon Negation Processing Time *vs* Complexity

	DATA	INHERENT C	GRID C	GRID T	RLE C	RLE T	POLY C	POLY T
1	1A	.292	.350	35.400	6.997	.340	4.600	1.500
2	2A	2.959	1.781	38.920	20.795	1.880	18.900	4.980
3	3A	8.951	5.164	47.320	35.494	5.340	30.300	7.040
4	4A	20.387	7.307	52.680	65.200	11.660	58.400	14.200
5	5A	70.614	20.841	86.080	120.187	26.400	101.400	22.620
6	6A	170.056	65.197	196.360	235.213	55.140	190.600	33.460
7	7A	378.875	86.825	249.040	353.997	90.240	269.100	52.700

Table 4-15: Enveloping Processing Times for Grids, RLE's and Polygons

and inherent data complexity on the right. The correlation values and the difference between them are:

- correlation using representation complexity: $r^2 = 0.961$,
- correlation using inherent data complexity: $r^2 = 0.972$,
- difference of correlations $\Delta r^2 = -0.011$.

4.5.3.4 Enveloping

The test results for the enveloping algorithms for the different representations applied to the test data are summarized in Table 4-15.

Figure 4-23 shows two graphs of grid enveloping processing time (in seconds) against the two different complexity measures; representation complexity on the left

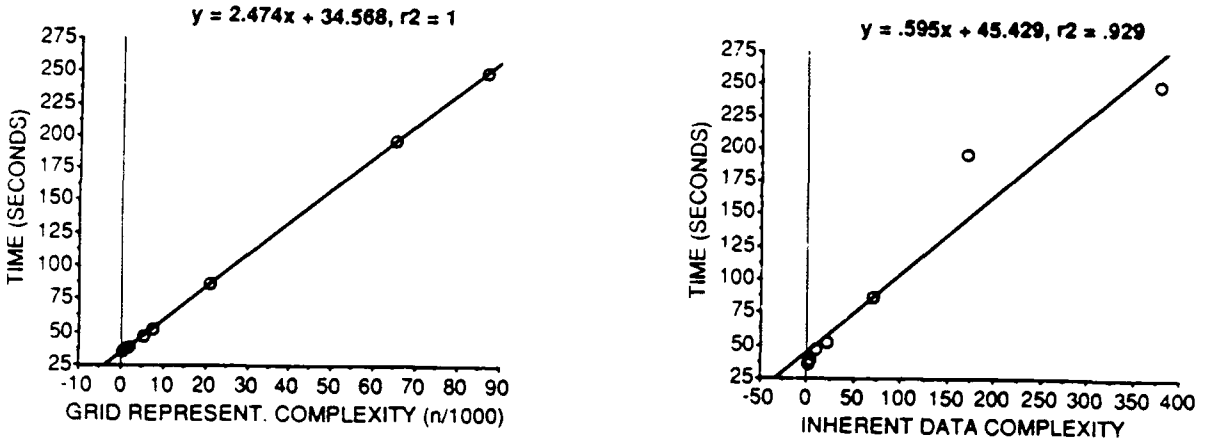


Figure 4-23: Grid Enveloping Processing Time *vs* Inherent Data Complexity

and inherent data complexity on the right. The correlation values and the difference between them are:

- correlation using representation complexity: $r^2 = 1.000$,
- correlation using inherent data complexity: $r^2 = 0.929$,
- difference of correlations $\Delta r^2 = 0.071$.

Figure 4-24 shows two graphs of RLE enveloping processing time (in seconds) against the two different complexity measures; representation complexity on the left and inherent data complexity on the right. The correlation values and the difference between them are:

- correlation using representation complexity: $r^2 = 0.997$,
- correlation using inherent data complexity: $r^2 = 0.973$,
- difference of correlations $\Delta r^2 = 0.024$.

Figure 4-25 shows two graphs of polygon enveloping processing time (in seconds) against the two different complexity measures; representation complexity on the left and inherent data complexity on the right. The correlation values and the difference between them are:

- correlation using representation complexity: $r^2 = 0.989$,
- correlation using inherent data complexity: $r^2 = 0.936$,
- difference of correlations $\Delta r^2 = 0.053$.

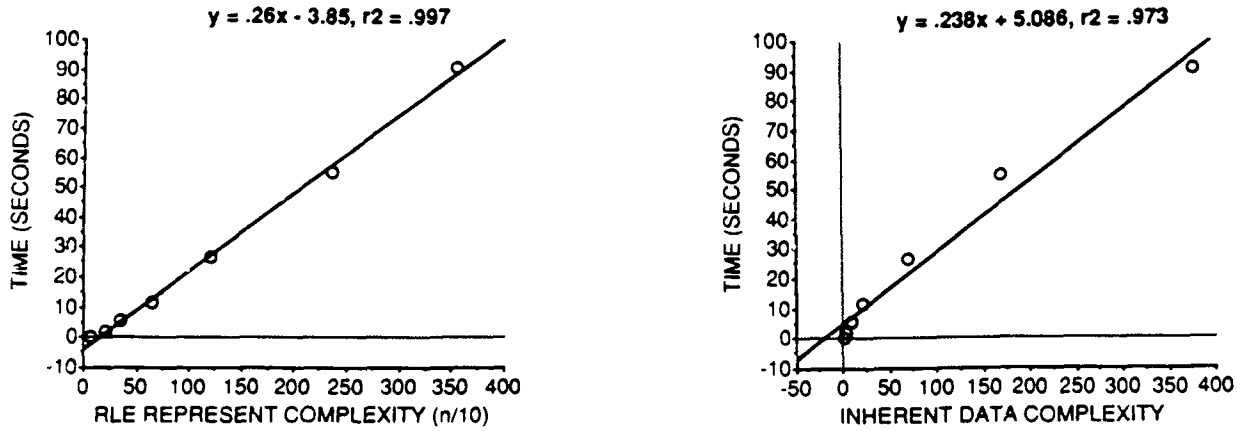


Figure 4-24: RLE Enveloping Processing Time *vs* Inherent Data Complexity

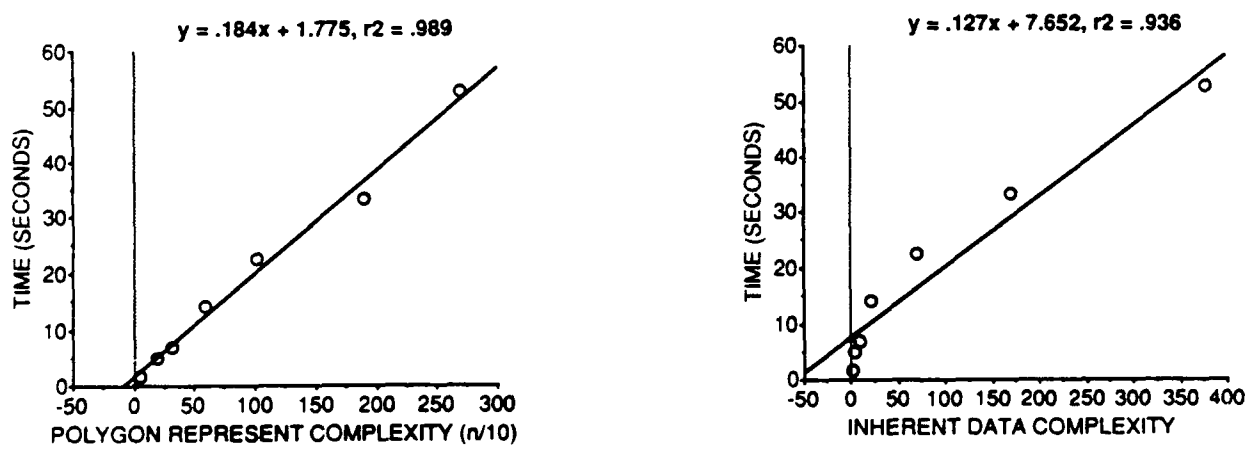


Figure 4-25: Polygon Enveloping Processing Time *vs* Inherent Data Complexity

Method	*Performance Model*	*Correlation*
Grid Intersect	$y = .013x + 85.081$	$r^2 = .728$
RLE Intersect	$y = .005x + .296$	$r^2 = .766$
Polygon Intersect	$y = -28.7 + 1.608x + .041x^2$	$r^2 = .985$

Table 4-16: Performance Models of Intersection Algorithms for Grids, RLE, and Polygons

Method	*Performance Model*	*Correlation*
Grid Union	$y = -.01x + 102.913$	$r^2 = .626$
RLE Union	$y = .009x + .51$	$r^2 = .804$
Polygon Union	$y = -108.299 + 7.128x + .025x^2$	$r^2 = .968$

Table 4-17: Performance Models of Union Algorithms for Grids, RLE, and Polygons

4.5.3.5 Summary of Performance Models

Tables 4-16--4-19 summarize the regression equations generated in the above tests. An analysis of the Δr^2 values shows that inherent data complexity is roughly equivalent to representation complexity in predictive value.

4.6 Analysis of Performance Study

The above study provides an answer to the questions posed in Section 2.3. The results indicate that:

- It is possible to define inherent data complexity for regions,
- The complexity measure correlates well with representation complexity,
- Performance models generated using the complexity measure are nearly as good as those generated using representation complexity.

From these results we specify a methodology for generating performance models:

1. define and implement algorithm.

Method	*Performance Model*	*Correlation*
Grid Negate	$y = -.001x + 87.449$	$r^2 = .419$
RLE Negate	$y = .003x + .263$	$r^2 = .935$
Polygon Negate	$y = .002x + .294$	$r^2 = .972$

Table 4-18: Performance Models of Negation Algorithms for Grids, RLE, and Polygons

Method	*Performance Model*	*Correlation*
Grid Envelope	$y = .595x + 45.429$	$r^2 = .929$
RLE Envelope	$y = .238x + 5.086$	$r^2 = .973$
Polygon Envelope	$y = .127x + 7.652$	$r^2 = .936$

Table 4-19: Performance Models of Enveloping Algorithms for Grids, RLE, and Polygons

2. analyze complexity of algorithm,

3. select test data for algorithm,

4. compute inherent complexity of test data,

5. run tests of algorithm in a baselined testbed environment,

6. perform a regression of processing time against inherent data complexity.

The following chapter will explore how the resulting performance models and the data complexity measure can become part of a testbed for spatial data processing.

5. Design of a Future Testbed

5.1 Purpose

The performance study of the previous chapter provides the basis for a terrain reasoning *testbed.* Such a testbed would:

- provide an infrastructure for terrain analysis applications,
- hide the details of spatial representations and relations,
- automatically optimize spatial operations,
- provide an environment for experimentation and development of spatial operations.

By distinguishing spatial semantics from internal representation, the testbed frees the users from the computer science details, and allows them to concentrate on representing domain knowledge.

5.2 Key Concepts

The primitive elements of the testbed are:

- representation,
- generic region,
- algorithm,
- method,
- generic method,
- plan.

Each of these testbed components, which are described below, are objects in an *object-oriented* paradigm.

Representation simply refers to the particular representation of spatial data; for example grid, RLE and polygon. A given dataset can be *instantiated* in any or all of these representations, and conversions exist from one representation to another. Within the testbed, the representation of a region is a computer science level detail which is hidden from the user.

Instead of talking about particular representations, the user talks about *generic regions.* The purpose of the testbed is to allow the user to perform some spatial operation on generic regions of possibly different representations without having to know about the representations or algorithms used. Currently, only generic regions are defined, but other spatial objects like curves or volumes could be similarly defined. A generic region is defined as follows:

```
;;;
;;; Generic Region Object
;;;
(defclass generic-region ()
  (:external
   :grid
   :rle
   :polygon
   :complexity))
```

A generic region represents a particular region in one or more ways. It may have some external definition in a file, or it may have one or more instantiated representations of the region. The complexity of the region object is also stored with the generic region; this is computed using the inherent data complexity measure defined in Chapter 4. Each of these properties of a generic region can be computed on an as-needed basis. New representations can be easily added to expand the definition of a generic region.

Algorithms are simply pieces of software which compute some spatial operation given input in some particular representation. The testbed is designed to isolate the user from having to know what representations are instantiated, and what algorithms apply to those representations. The algorithms available in the testbed are described in Chapter 3. New algorithms are easily added, and may be coded in Lisp or C.

A method is more abstract than an algorithm but is still dependent on the input representations. Methods capture information about a particular algorithm and make that available for higher level processes. A method is defined as follows:

```
;;;
;;; Spatial Method Definition
;;;
(defclass method ()
   :input-types
   :performance-model
   :output-type
   :output-complexity-estimation
   :algorithm)
```

A method takes generic regions as input. The method specifies which of the representations of the generic region are required for the algorithm as well as the resultant output representation. The performance model for the algorithm (see previous Chapter) is stored with the method. This is a function which, when applied to the input data, estimates the cost of executing the algorithm for the given input data.. In the future it would be desirable to have a function which would estimate the complexity of the output from the complexities of the input. When a method is called with generic regions as input, it returns a list of the following:

- the required input types,
- the estimated cost for running the algorithm on the data,

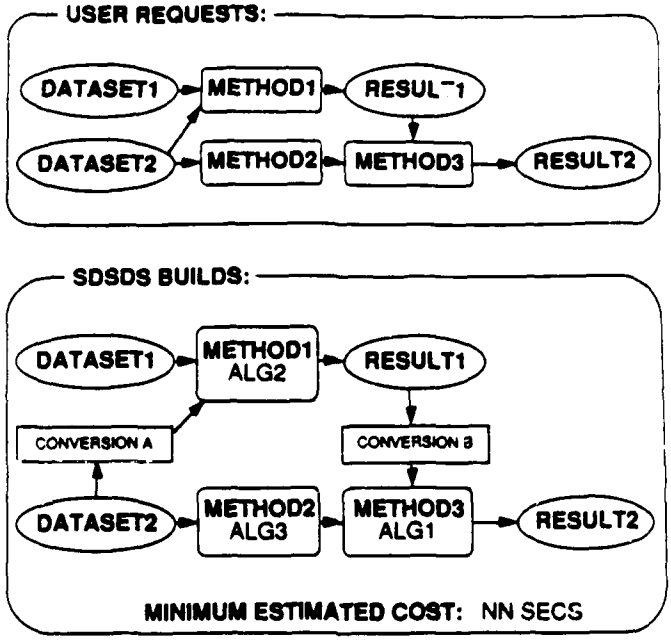


Figure 5-1: Example Plan

- a Lisp expression which, when evaluated, will apply the function to the data,
- the output type,
- the estimated output complexity (if available).

The *generic method* is the highest level of data abstraction. It defines a representation independent spatial operation; for example, enveloping. The generic method essentially contains a list of methods defined above. When a generic method is called with generic regions as input, it calls all appropriate methods and returns the list of results. This result contains all the information about how the operation could be computed and at what cost. What has not been taken into account is that the generic regions may not have all possible representations instantiated. Hence the computation of some of the methods returned by the generic method will require update operations to be performed on the generic region to instantiate those representations specified as required input types. This is the level at which planning takes place.

A *plan* is a sequence of one or more methods which takes the specified input generic regions and produces a generic region with a specific instantiated representation in an optimal way (Figure 5-1). The planner takes as inputs: input generic regions, a sequence of one or more generic methods to be performed, and a desired output type. Each generic method is expanded as described above. Conceptually, a graph is then built with arcs from the input to the output through nodes corresponding to each of these method expansions. Additional nodes are introduced between method nodes to account for data conversion if the output of one function does not match the input of the next. The path with the minimum total estimated cost, including the cost of data conversion, is chosen as the *plan* for implementing the sequence of generic methods.

Name	*Inherent Data Complexity*	*Representation*
3B	9.846	Polygon
5B	97.827	Grid

Table 5-1: Example Regions

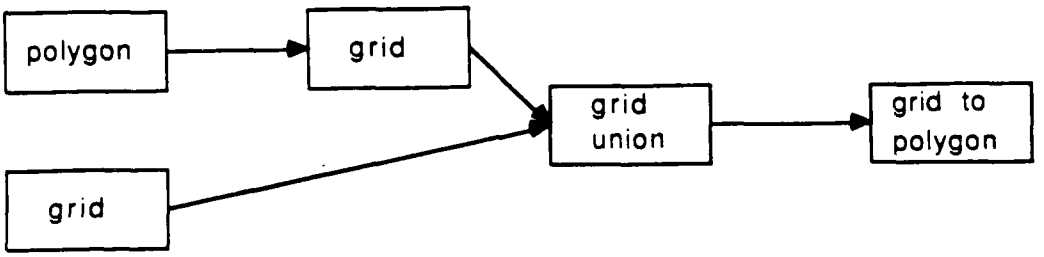


Figure 5-2: Plan 1: Polygon Union

For example, assume that a terrain analyst has the two regions in the following shown in Table 5.2. The analyst's goal is to find the union of 3B and 5B in polygonal form, and to do so in the least amount of time. Given the representations and algorithms developed in this study, the analyst has two obvious solutions. One is to convert 3B from polygon to grid, perform a grid union, and convert the answer to polygonal form (Figure 5-2). The second is to convert 5B from grid to polygon, and perform a polygon union operation (Figure 5-3). There is, however, a third, less obvious solution: convert 3B and 5B to RLE, use RLE union, and convert the result to polygon (Figure 5-4).

An automatic planner could now use the performance models for these algorithms, and the complexity of the data to evaluate each of the plans and choose the best. There is a problem, however, in computing the cost of plans 1 and 3. We currently have no way to estimate the complexity of the output of an algorithm from the complexity of the input. What is needed, in addition to the performance model

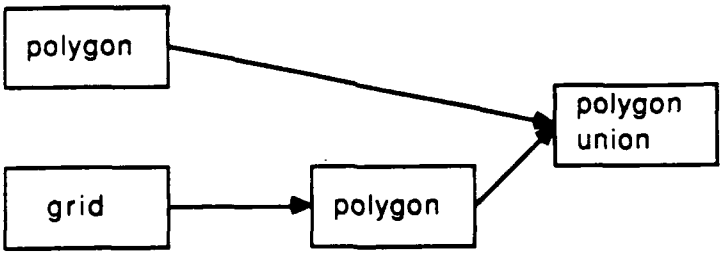


Figure 5-3: Plan 2: Polygon Union

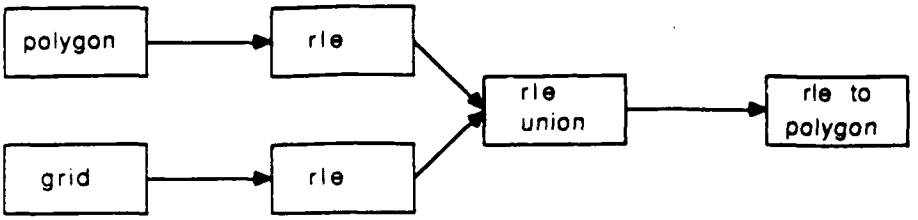


Figure 5-4: Plan 3: RLE Union

Plan	*Initial Conversions*	*Union Operation*	*Output Conversion*	*Total Time*
Plan 1	66.789	101.836	204.763	373.397
Plan 2	192.643	948.942	0.0	1141.585
Plan 3	183.345	1.479	182.353	367.177

Table 5-2: Plan Processing Times

for each algorithm, is a function which estimates the complexity of the output from the complexity of the input. This is an area for future study. The alternatives are either to test the complexity of the output (which is time consuming) or to make conservative assumptions about the expected output. For example, in the case of union, we can assume that the complexity of the output will be less than or equal to the sum of the complexities of the inputs.

Using this assumption completes the cost prediction of each of the plans. The Table 5.2 shows the predicted costs (in seconds) for each stage of the plans, as well as the total cost of the plan: This table indicates that plan 3, in which both objects were converted to RLE's, has the lowest predicted cost even though it involves the greatest number of conversion operations. Plan 3 is, in fact, much better than plan 1 which involves the fewest number of conversions. This clearly illustrates the value of performance models and data complexity measures in the optimization of spatial operations. A human analyst might choose an obvious, but non-optimal, solution to a problem. With the aid of an automated planner, more informed predictions can be made, and more possible plans examined.

5.3 Initial Design

Figure 5-5 shows the functional architecture of the testbed which supports the planning operation. The *dataset object manager* maintains the instantiated generic regions. The *method object manager* similarly maintains the methods and generic methods which make use of an *algorithm library*. The *plan object manager* is responsible for the planning process. This involves expanding the generic methods and using update methods to meet the constraints imposed by the input data requirements of

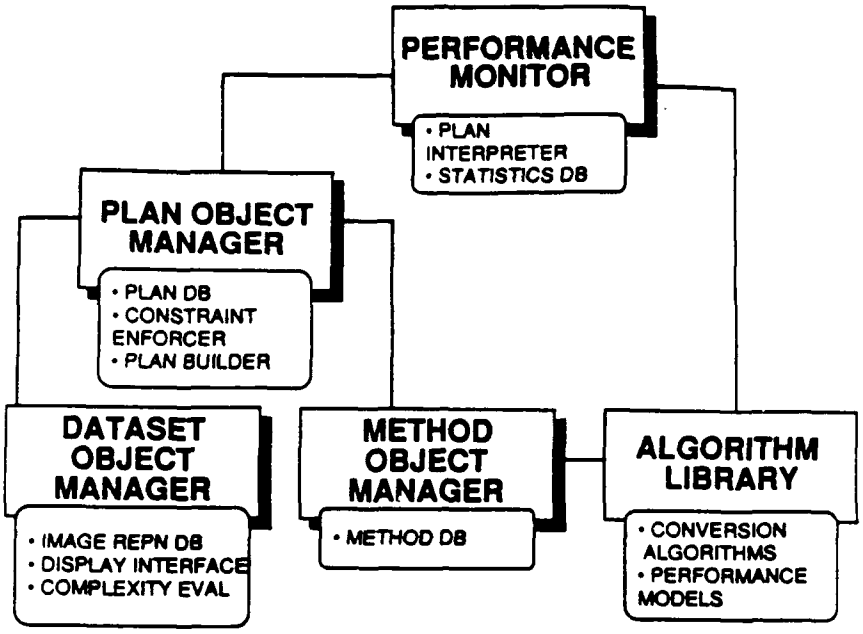


Figure 5-5: Functional Architecture

the various algorithms. This functional component also selects the best path, which becomes the plan which is executed by the *performance monitor*. The performance monitor can optionally evaluate the performance of the plan or subportions of it.

The current version of the testbed was assembled on the hardware architecture shown in Figure 5-6. The software architecture of the testbed is illustrated in Figure 5-7. The testbed software was rapidly prototyped to aid the testbed design process. The specification of the user interface helps to clarify the functionality of the testbed and the way in which users would interact with it. The initial user interface is described below.

The testbed interface is graphically oriented. One window is used for the display of region data, another for the display of performance statistics, and another for the graphical interaction with instantiated objects like generic regions, methods, and plans. Mouse sensitive buttons allow many functions to be performed by clicking on the screen.

The user gets data into the system by loading a dataset from file (Figure 5-8). This is done by clicking the mouse on the "Load Dataset" button and selecting one of the available datasets. When a dataset is instantiated, a generic region object is created (with no instantiated representations) and added to the dataset window.

These objects in the dataset window are mouse sensitive (Figure 5-9). Clicking the mouse on one of these objects produces a menu of possible operations that can be performed on that object.

The representations of a generic region can be instantiated (Figure 5-10) by clicking on the "Representation" button and then selecting a region object. The currently instantiated representations for a generic region are displayed in the generic region object. Representations can also be instantiated by using the update algorithms (Figure 5-11).

Generic methods are also objects which can be instantiated (Figure 5-12). As an

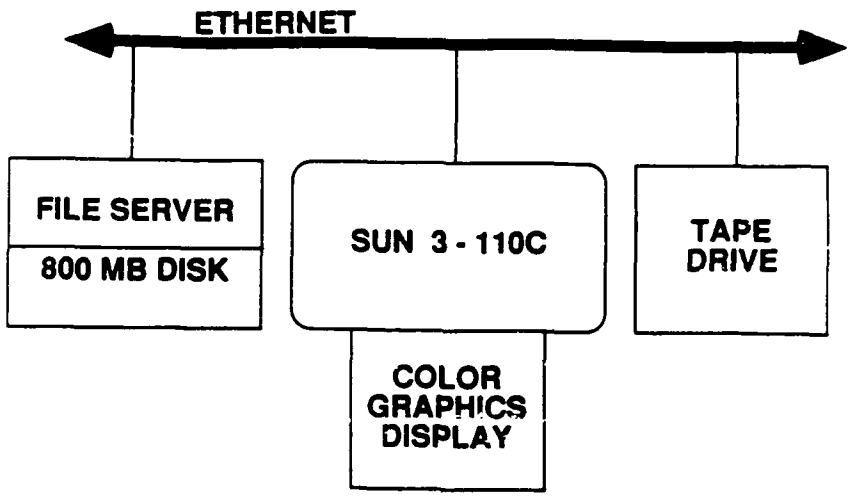


Figure 5-6: Hardware Architecture

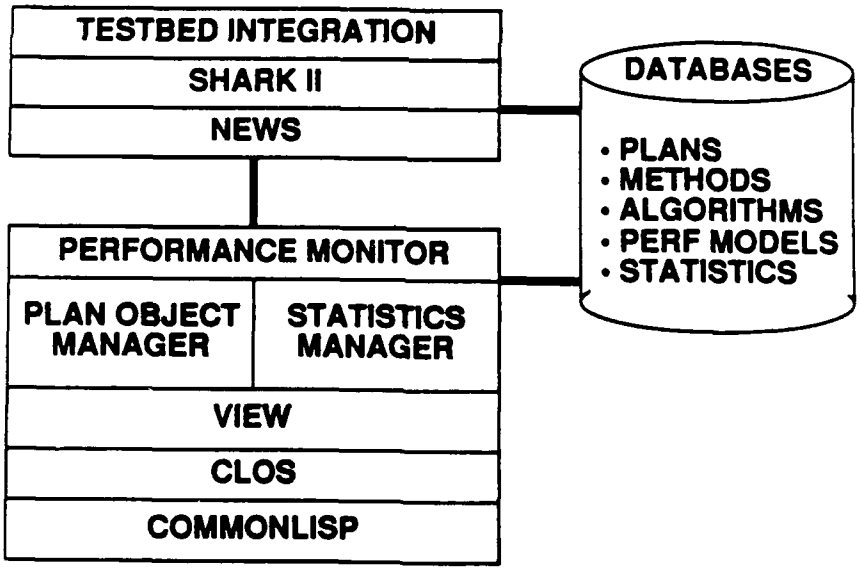


Figure 5-7: Software Architecture

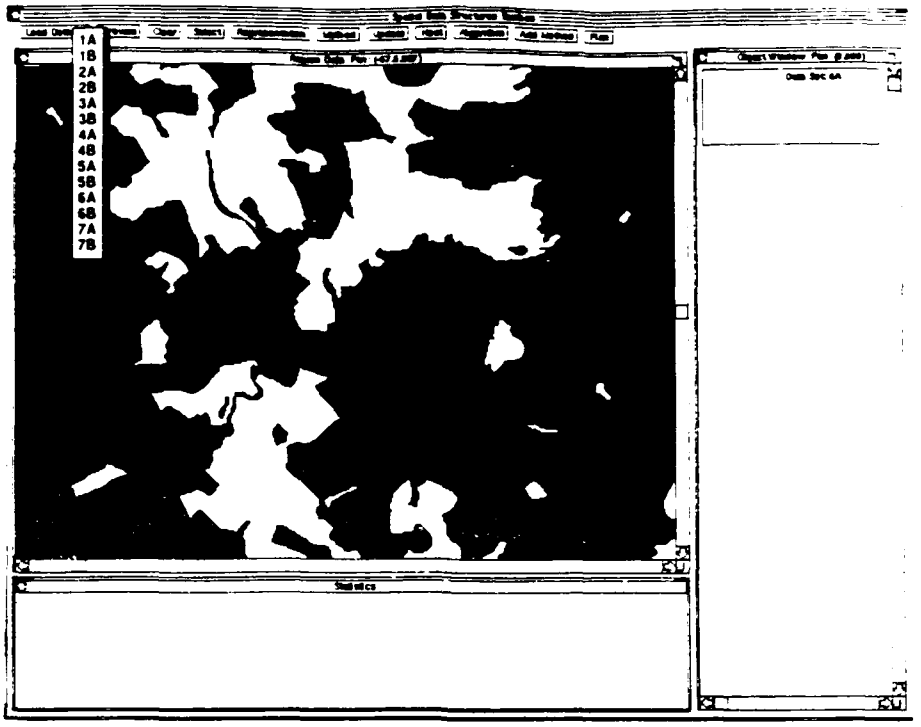


Figure 5-8: Instantiating Data Sets

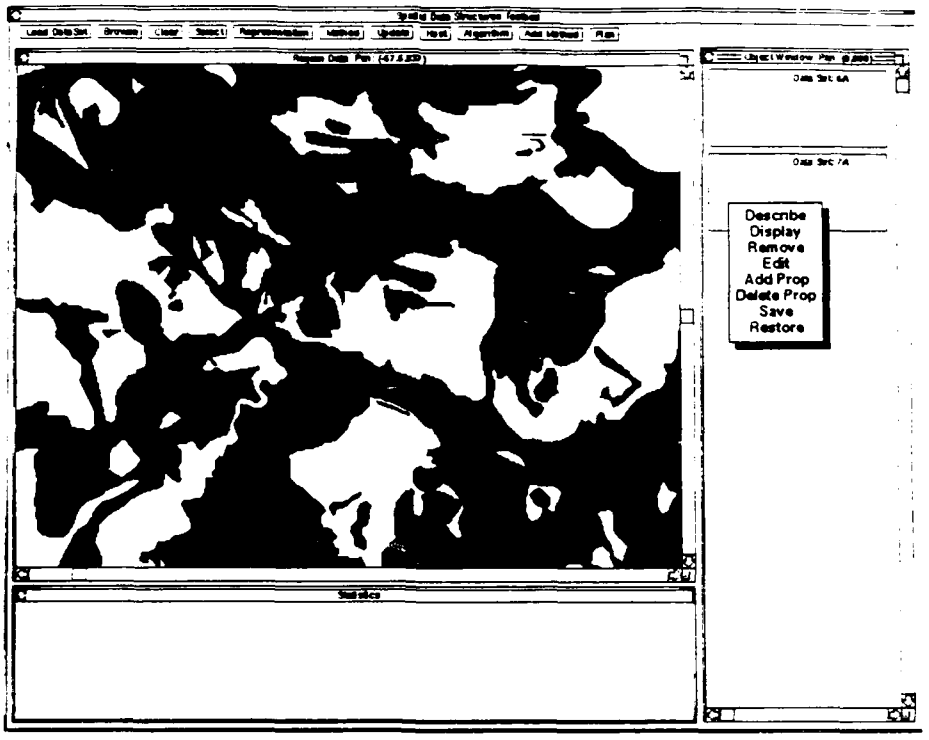


Figure 5-9: Generic Region Objects

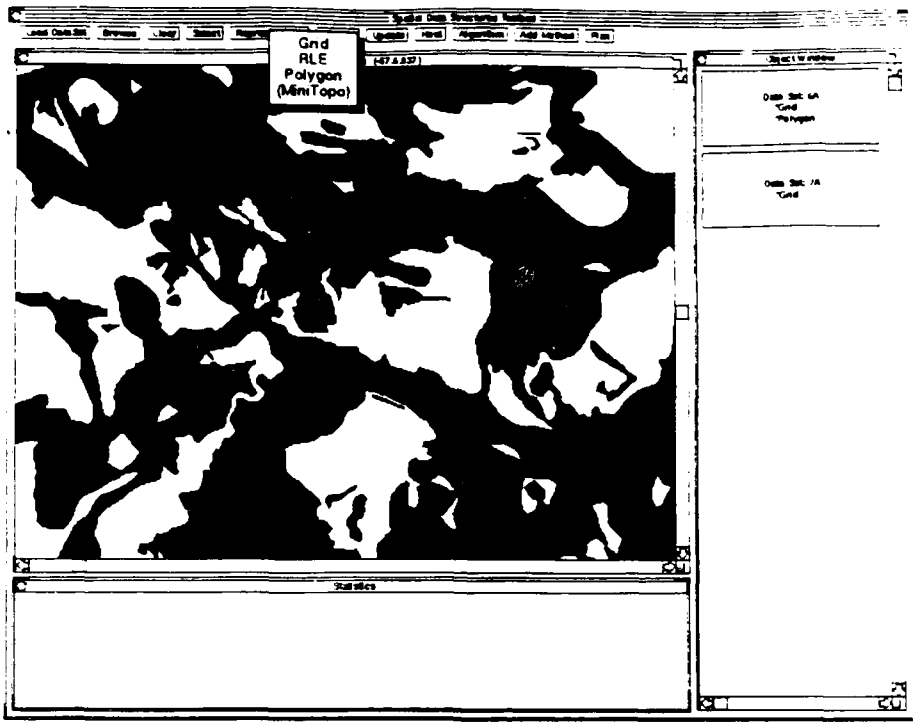


Figure 5-10: Instantiating Representations

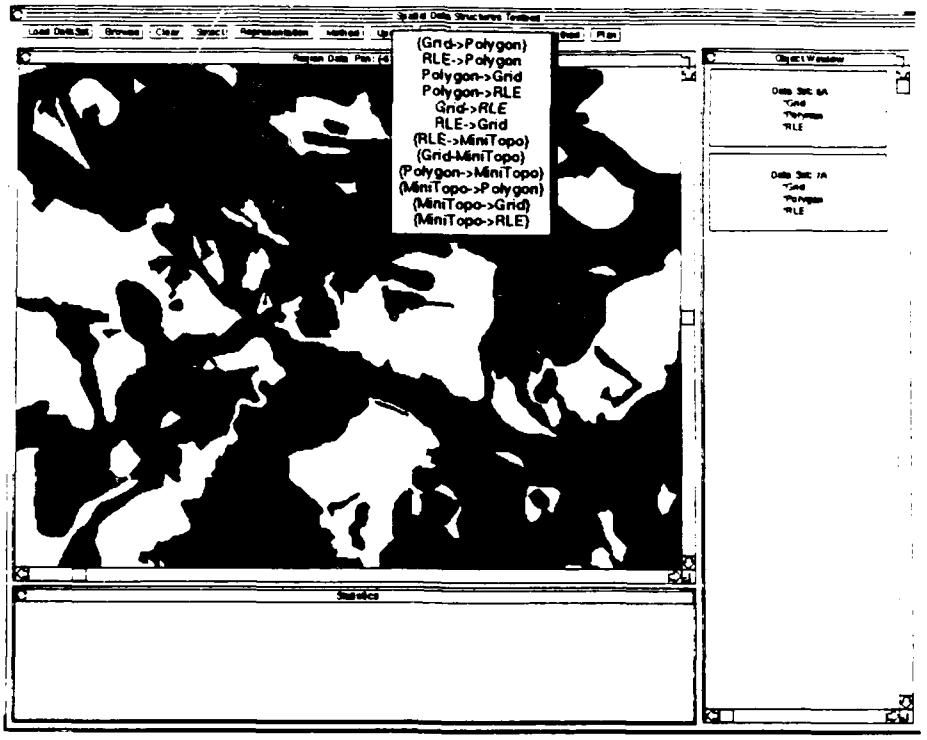


Figure 5-11: Representation Conversions

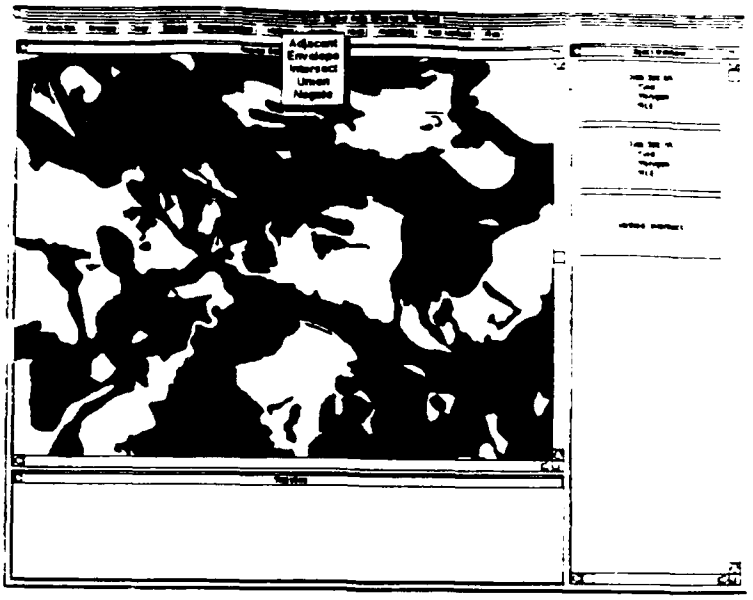


Figure 5-12: Processing Methods

experimentation tool the actual algorithms that make up a generic method may be chosen (Figure 5-13). This allows experimentation with results of planning given a restricted set of processing algorithms.

Plans are also objects (Figure 5-14) with specific inputs and a sequence of generic methods. An interactive plan builder would allow the user to graphically design a spatial operation by selecting methods like building blocks and connecting their inputs and outputs together. Such a facility has not been built.

After a plan has been executed, a new output object is generated and statistics can be calculated (Figure 5-15).

Another useful user interface feature of the testbed is the graphical database browser (Figure 5-16) This facility allows the user to examine many datasets at once. The datasets, which may be in memory or stored on a disk, can then be selected for further processing.

The complete design and implementation of the testbed is beyond the scope of this phase of the project. However, the preliminary design described above demonstrates the relationships among the fundamental concepts and substantiates the viability and value of the eventual testbed.

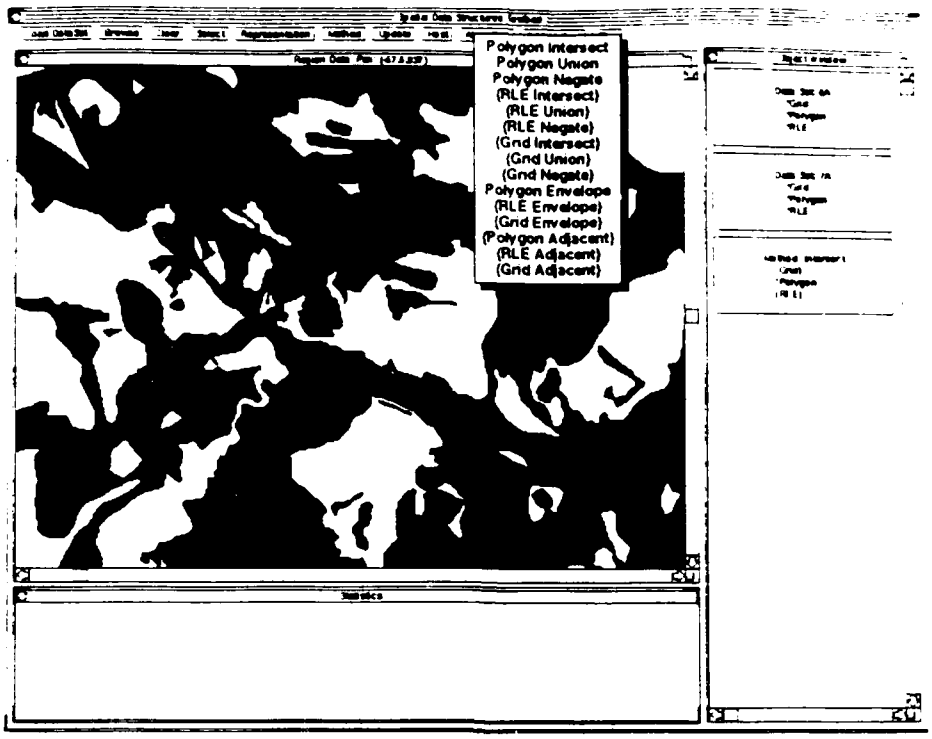
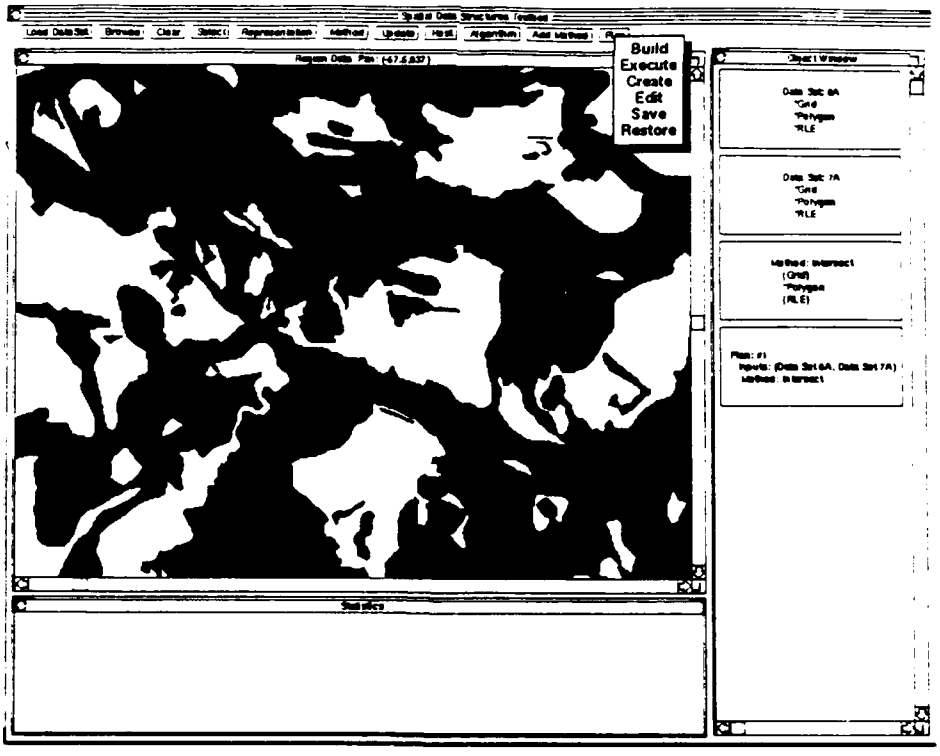


Figure 5-13: Processing Algorithms



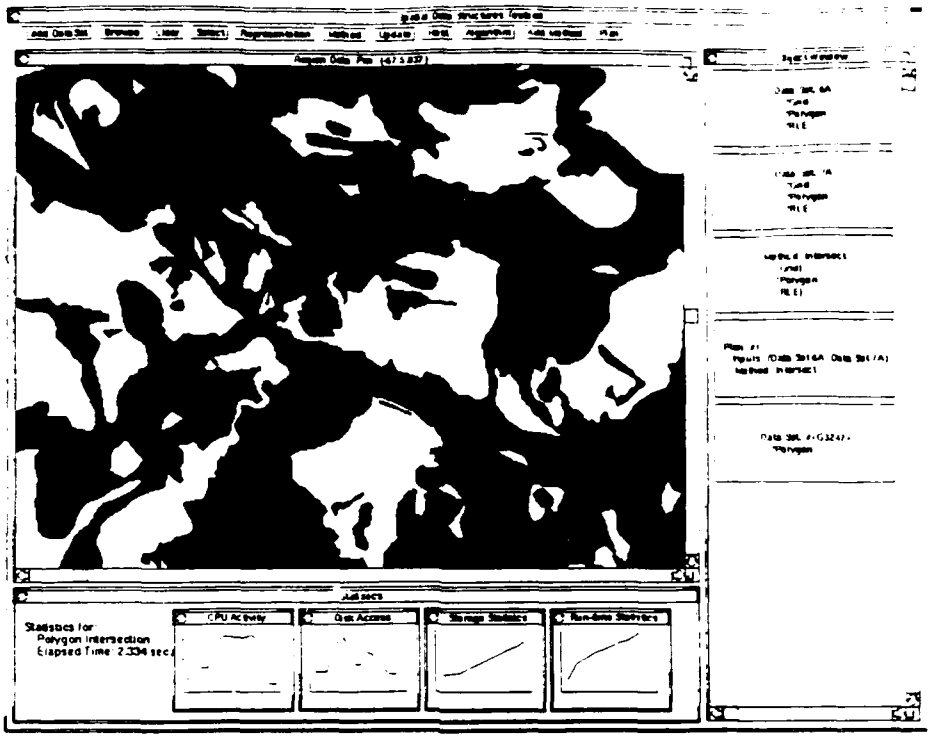


Figure 5-15: Plan Execution

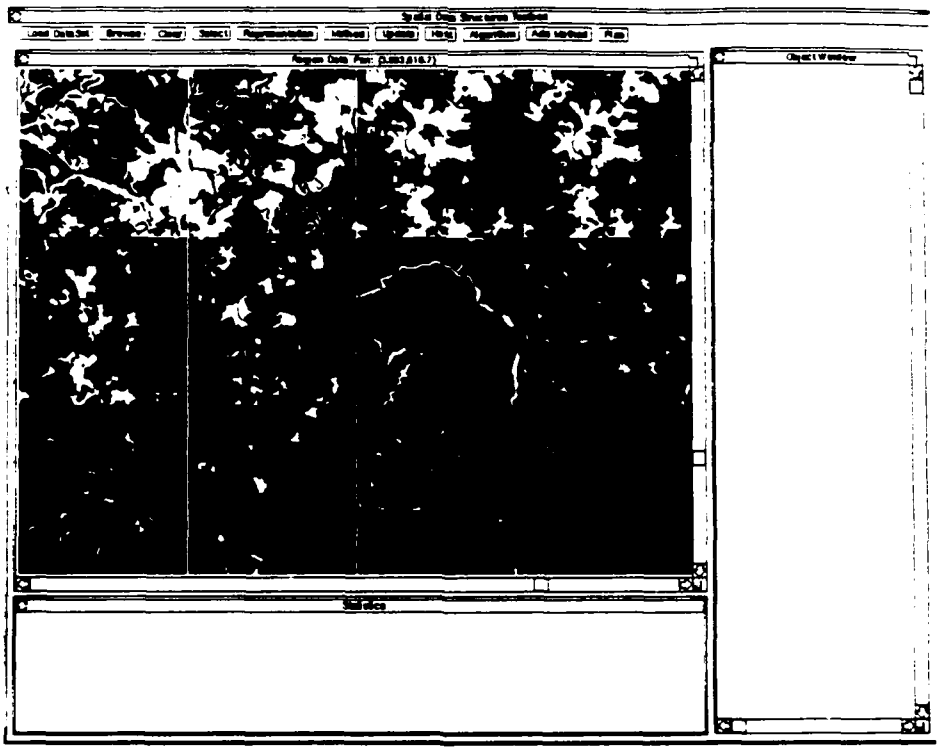


Figure 5-16: Data Set Browser

6. Conclusions

6.1 Summary of Study

The Phase II research began with the selection, implementation and evaluation of common terrain representations and operations. Data was chosen for experimentation, and the problem of expressing the complexity of data was researched. An inherent data complexity measure was defined, and tested on the chosen data. A test environment was built, and the algorithms were evaluation within this framework. The results of the tests were used to generate performance models based on both representation complexity and data complexity. The performance models were evaluated with respect to there predictive value. Using the notions of data complexity and performance modeling, an initial spatial analysis testbed design was proposed.

6.1.1 Resolved Issues

The major results of this Phase II research are:

- the object-oriented implementation of common terrain representations and spatial operations,
- the design of an effective inherent data complexity measure,
- the definition of representation independent performance modeling,
- the specification of a methodology for generating performance models of algorithms,
- the initial design for a spatial reasoning testbed using generic regions, generic methods, and plans,
- the definition of spatial operation optimization using performance models.

6.1.2 Unresolved Issues

The following are questions suggested by the Phase II study, and which are currently unanswered:

- What is computational expense of calculating inherent data complexity, and is the cost prohibitive in a run-time environment?
- What is the validity of the performance models in practice?
- How can the complexity of algorithm output be estimated from algorithm input?
- Can inherent data complexity be computed by a statistical sampling of the dataset?

6.2 Recommendations

This study provides the foundation for spatial reasoning testbed. The existence of such a testbed would have great value to terrain analysts and computer scientists working on automating terrain analysis or robot navigation.

To realize such a testbed, the following tasks are proposed:

- complete the testbed design,
- complete the implementation of a prototype testbed,
- given the current set of representations and algorithms, answer the unresolved issues within the testbed environment,
- expand the available set of representations and algorithms in the testbed using the methodology outlined in this research,
- evaluate the performance of the testbed.

7. Appendix A: Data Sets

The following pages contain the region data, extracted from a CATTS database, used for testing and evaluating data structures and algorithms.

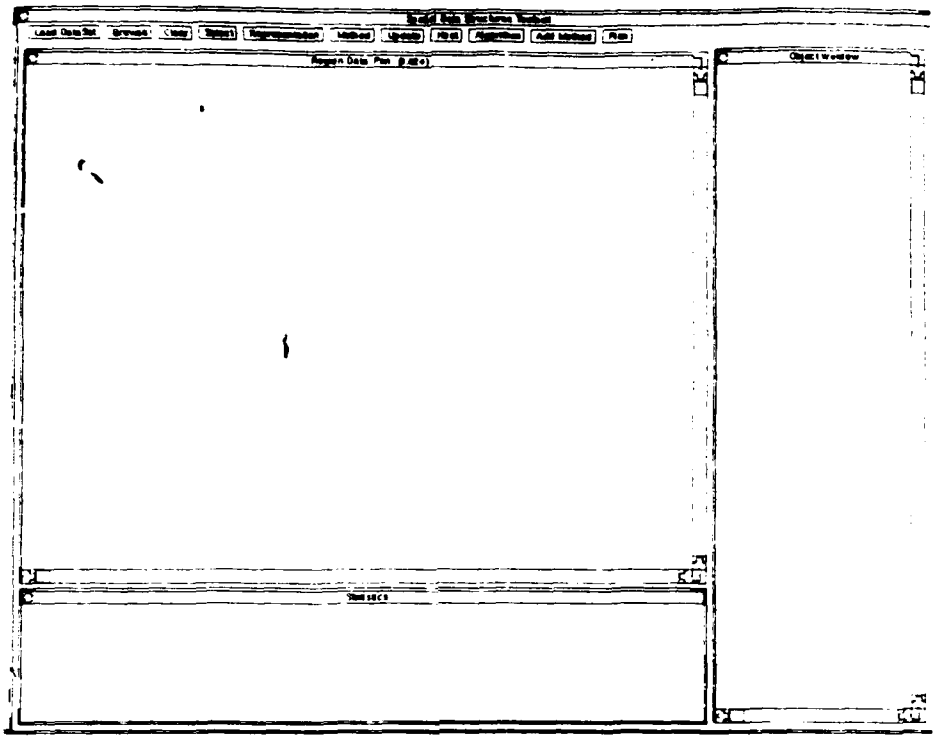


Figure 7-1: Dataset 1A.

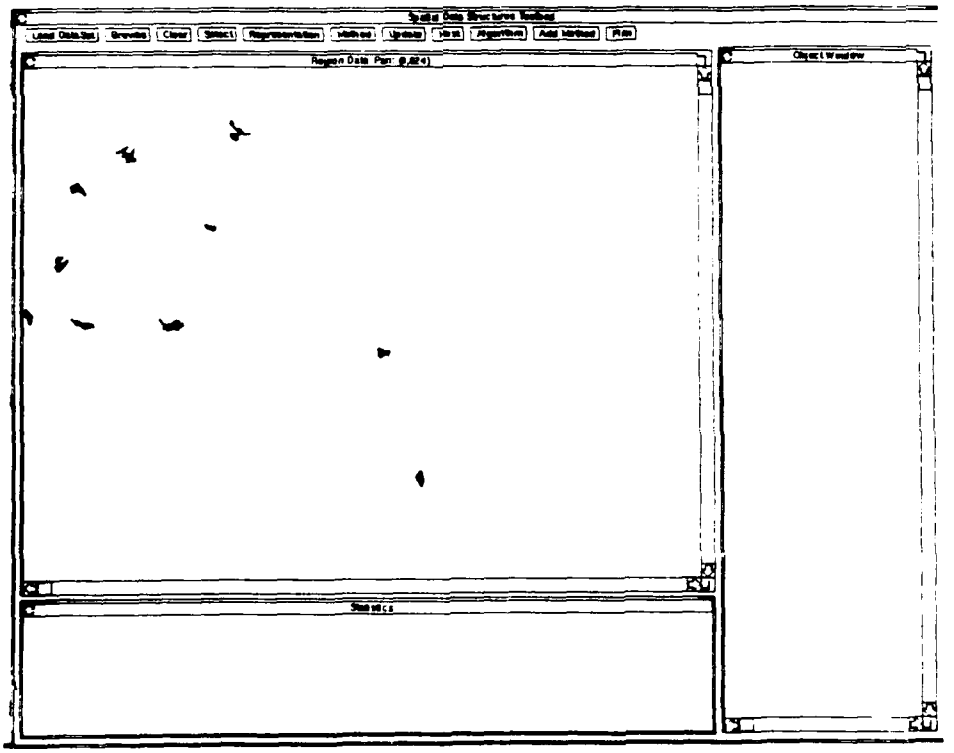


Figure 7-2: Dataset 2A.

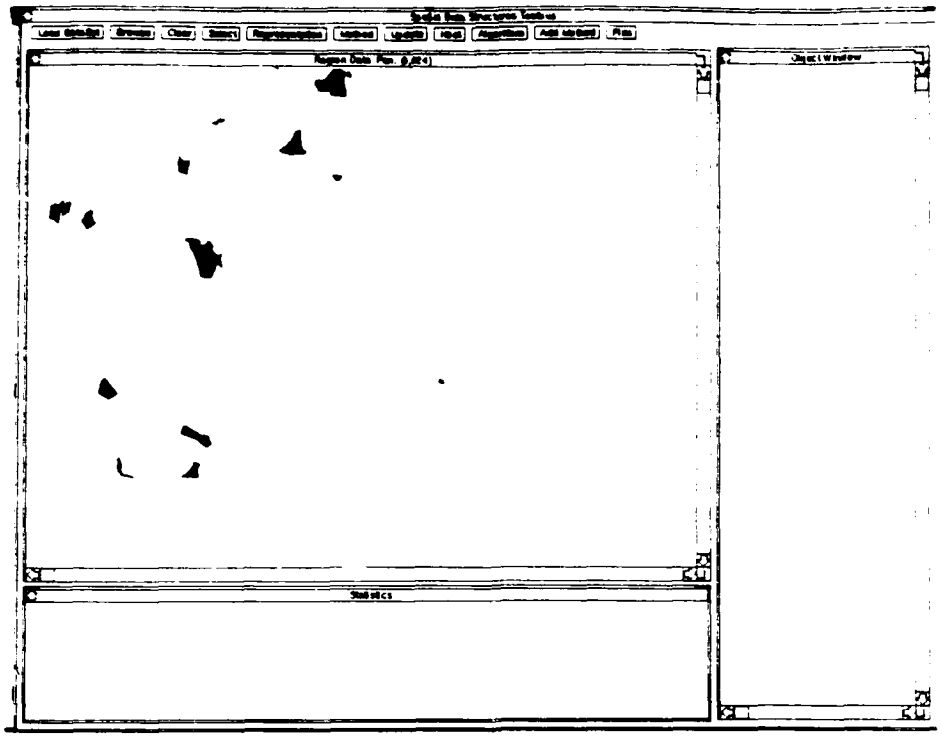


Figure 7-3: Dataset 3A.

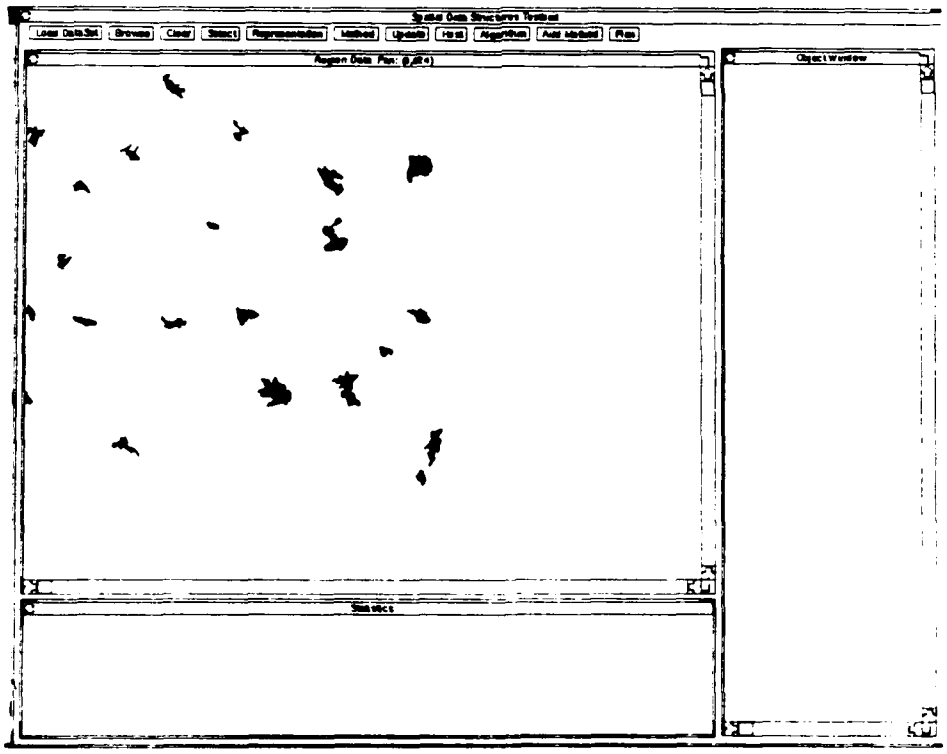


Figure 7-4: Dataset 4A.

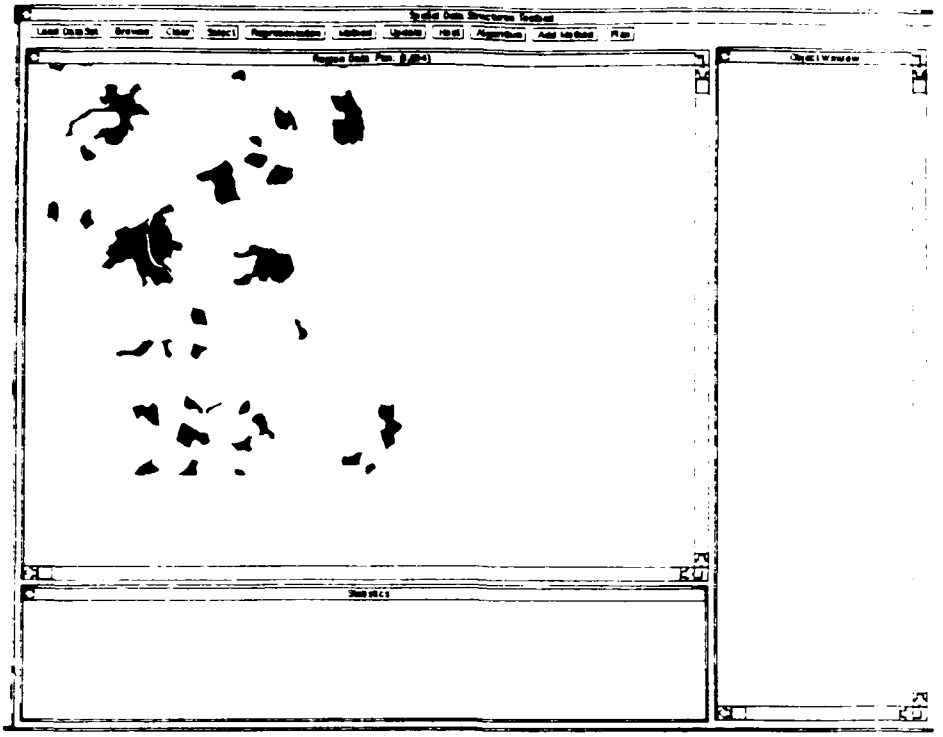


Figure 7-5: Dataset 5A.

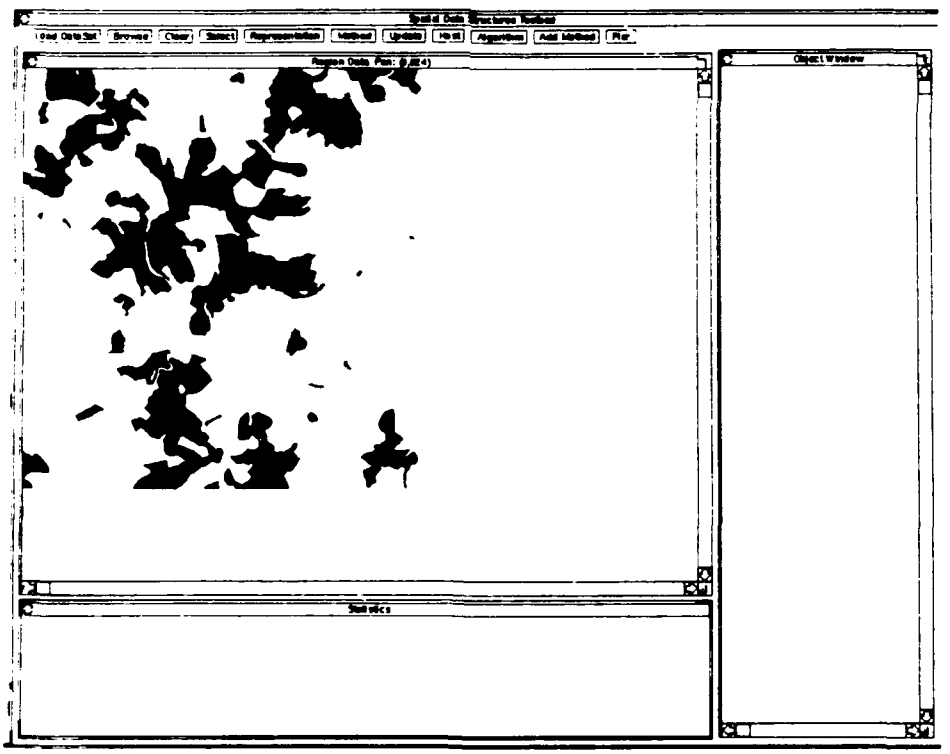


Figure 7-6: Dataset 6A.

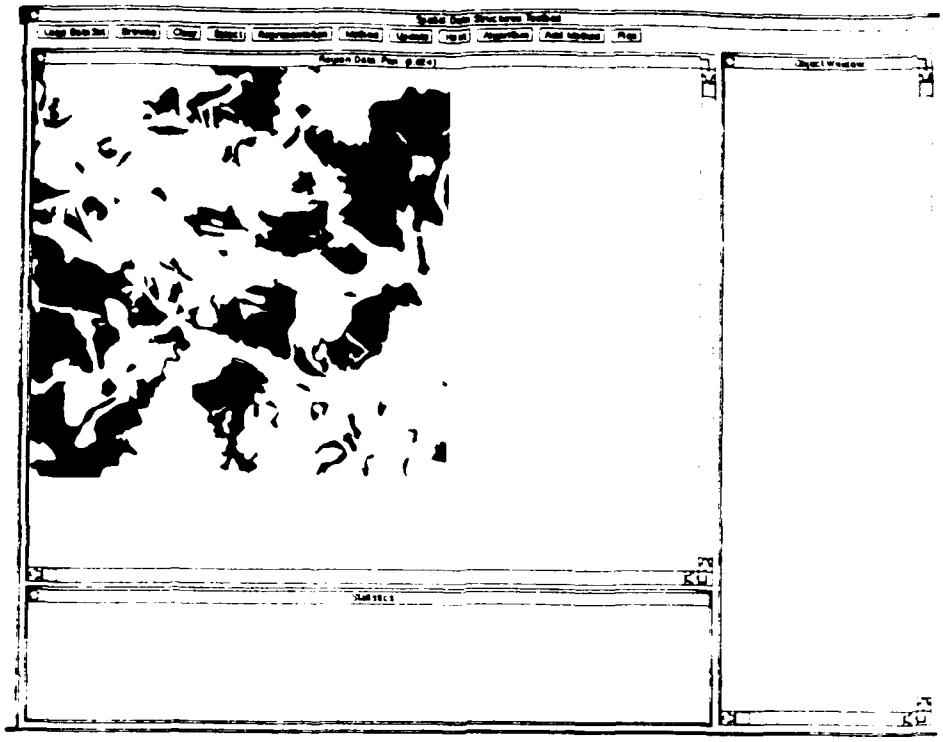


Figure 7-7: Dataset 7A.

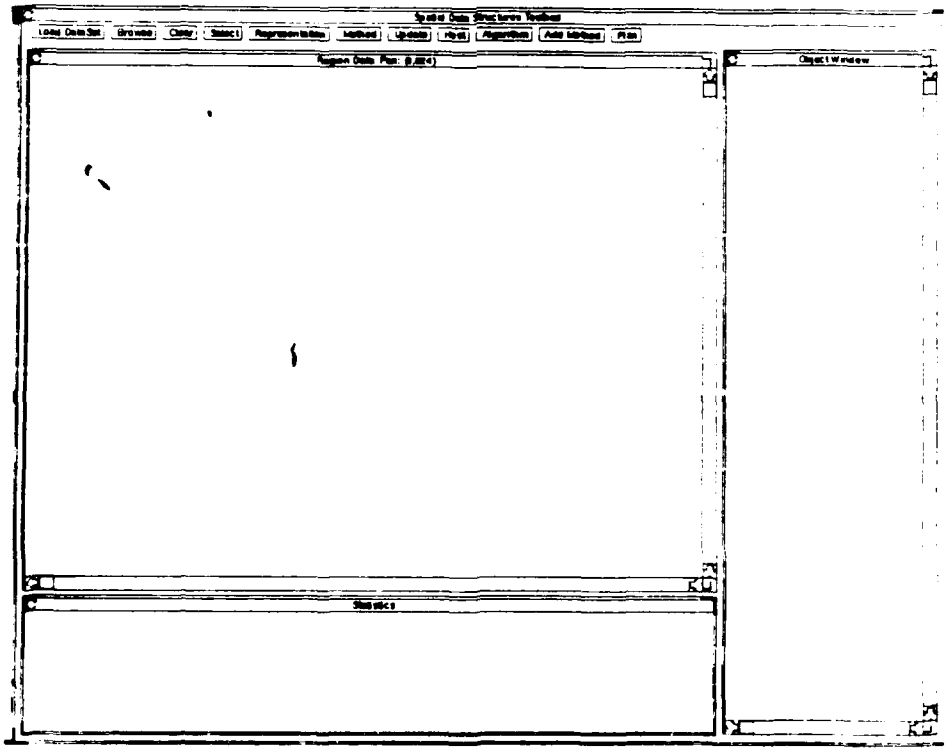


Figure 7-8: Dataset 1S: 1A shifted -3 pixel in X axis and 3 pixels on Y axis.

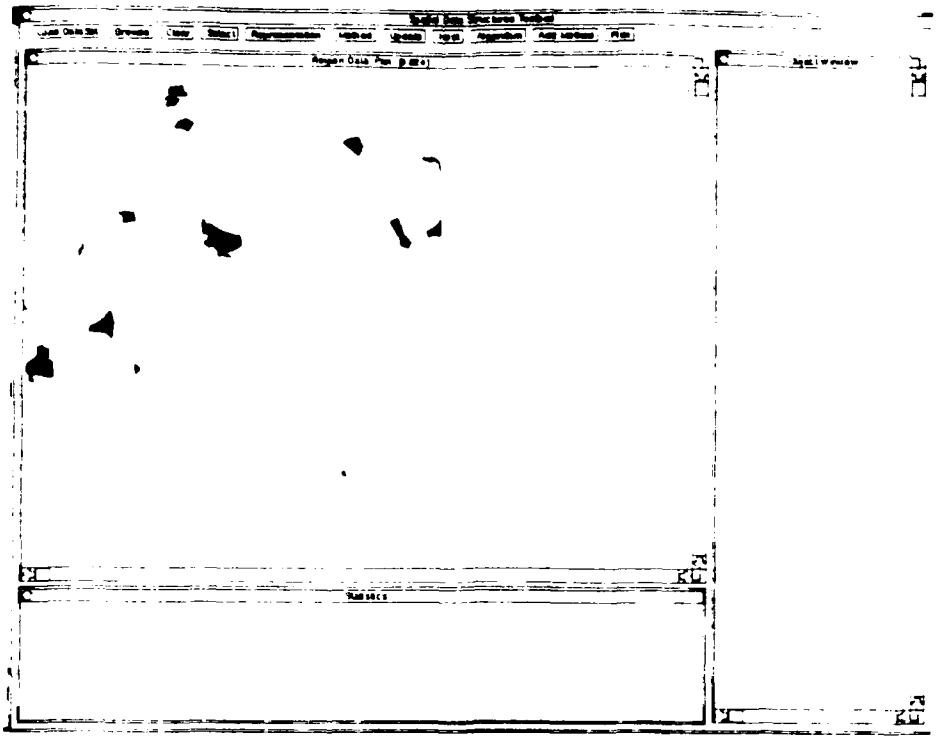


Figure 7-9: Dataset 3I: 3A with X and Y axis switched.

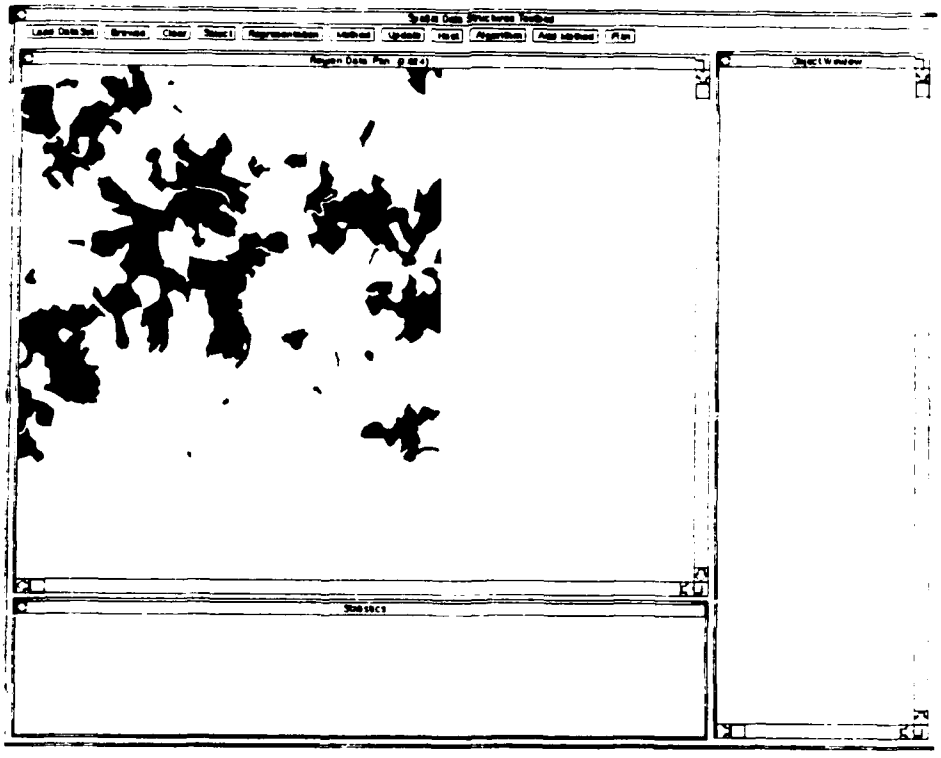


Figure 7-10: Dataset 6I: 6A with X and Y axis switched.

8. Appendix B: Results of Set Operations

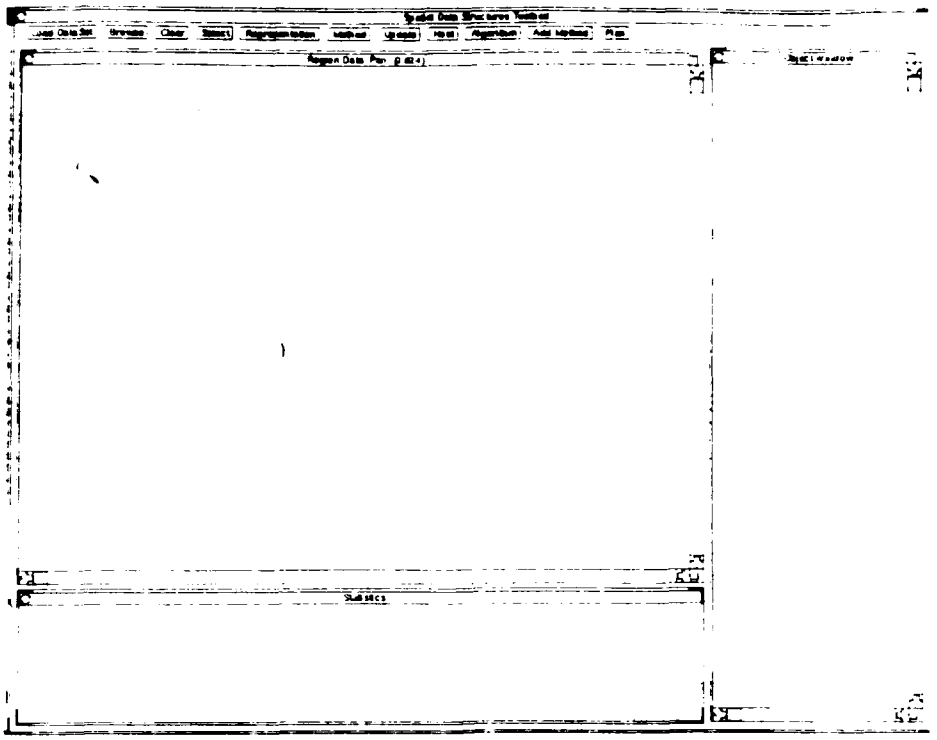


Figure 8-1: Intersection of Datasets 1A and 1S

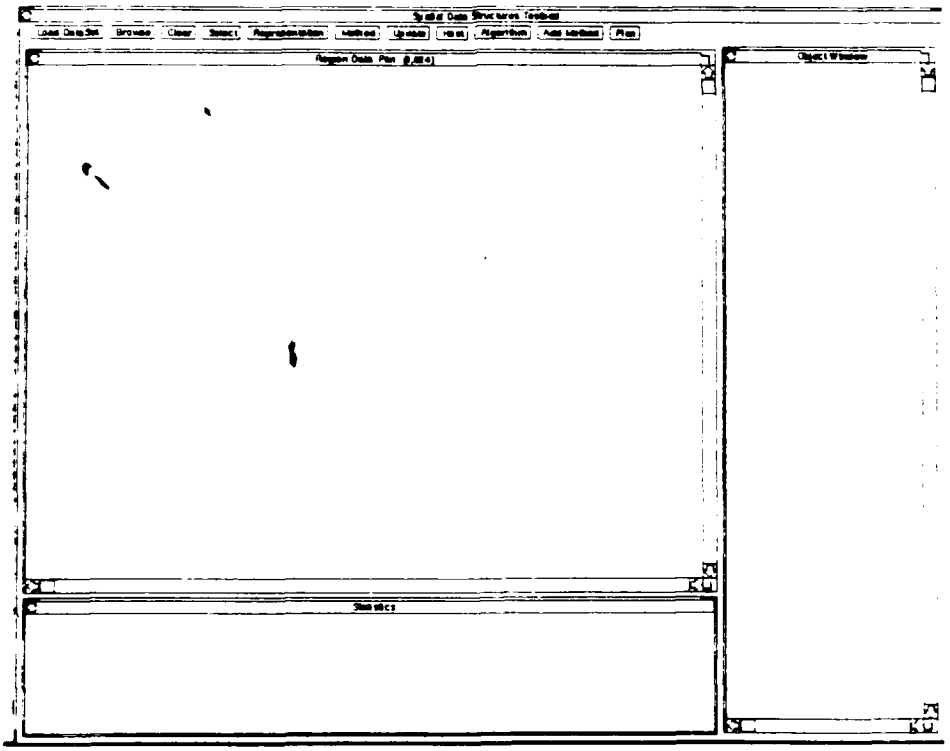


Figure 8-2: Union of Datasets 1A and 1S

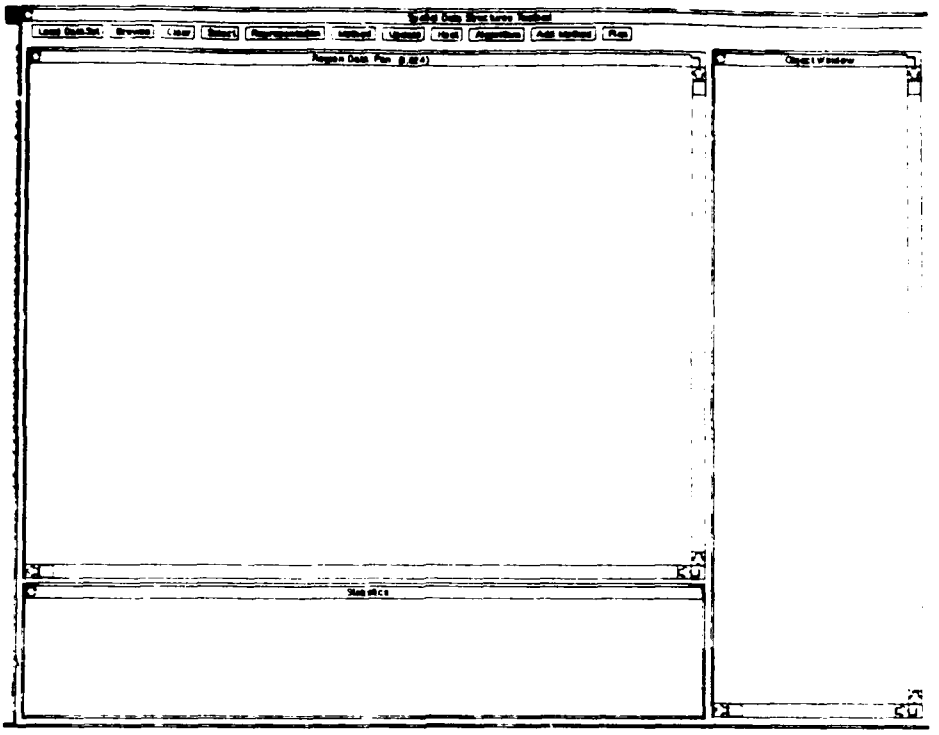


Figure 8-3: Intersection of Datasets 2A and 3I

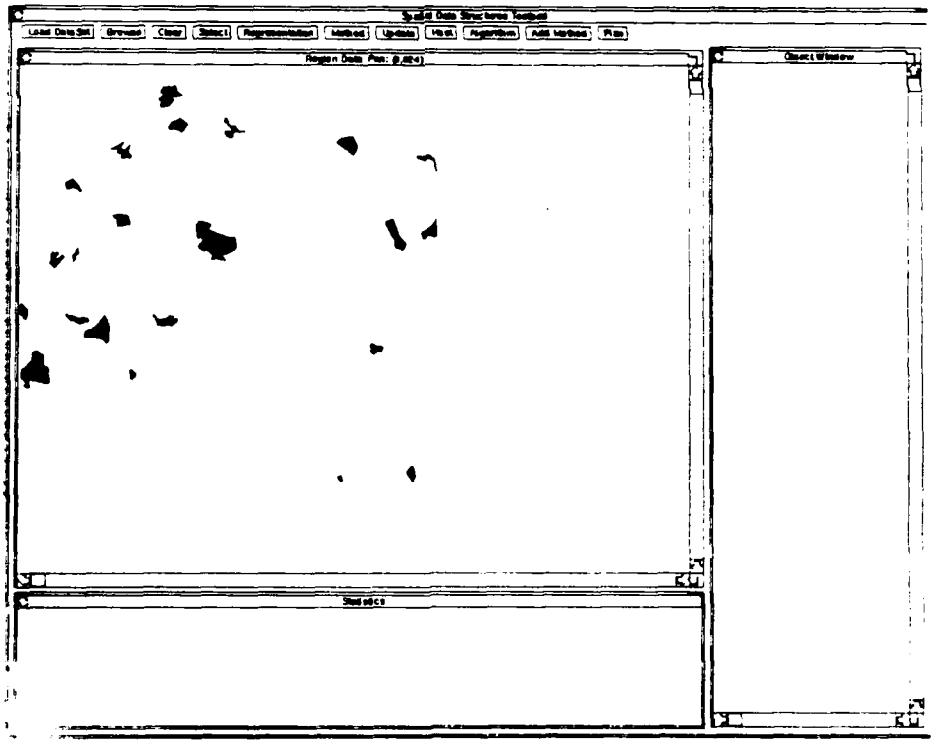


Figure 8-4: Union of Datasets 2A and 3I

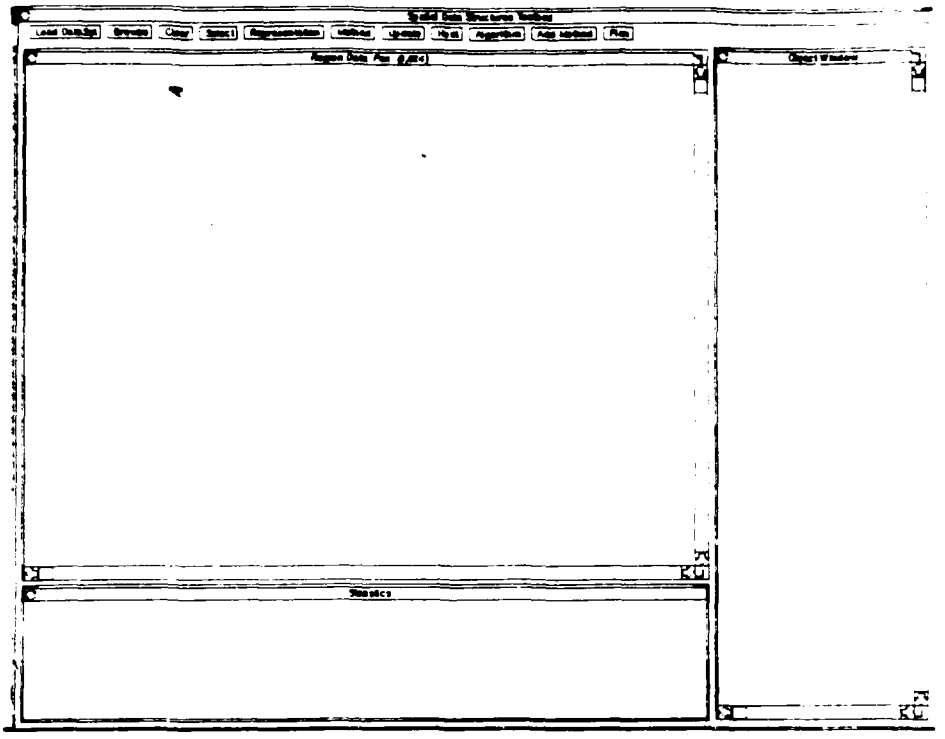


Figure 8-5: Intersection of Datasets 4A and 3I

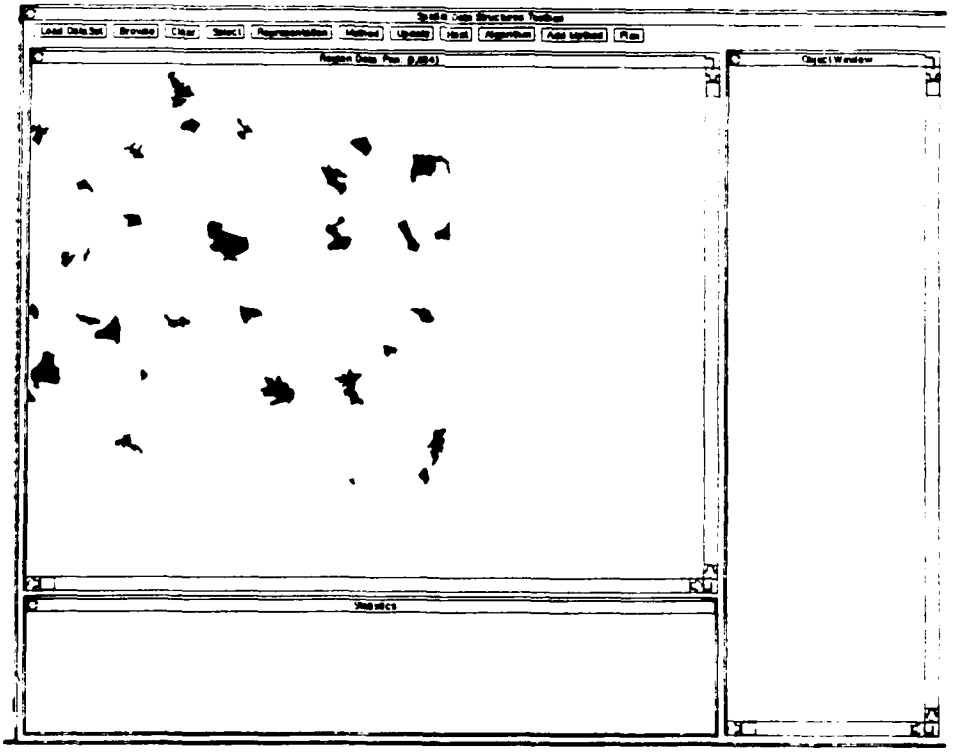


Figure 8-6: Union of Datasets 4A and 3I

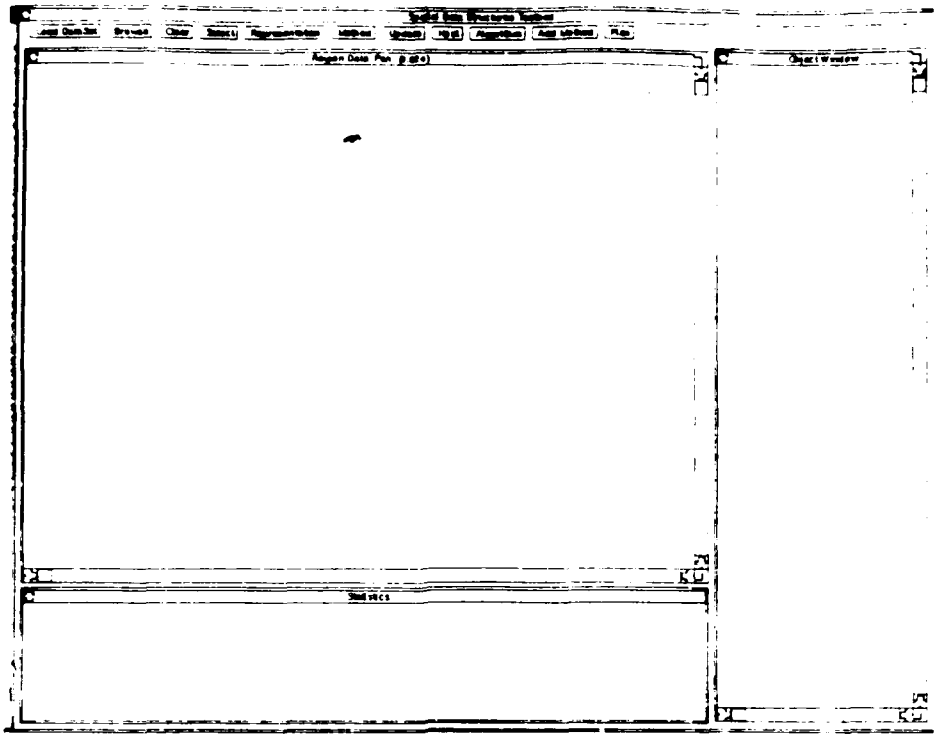


Figure 8-7: Intersection of Datasets 5A and 3I

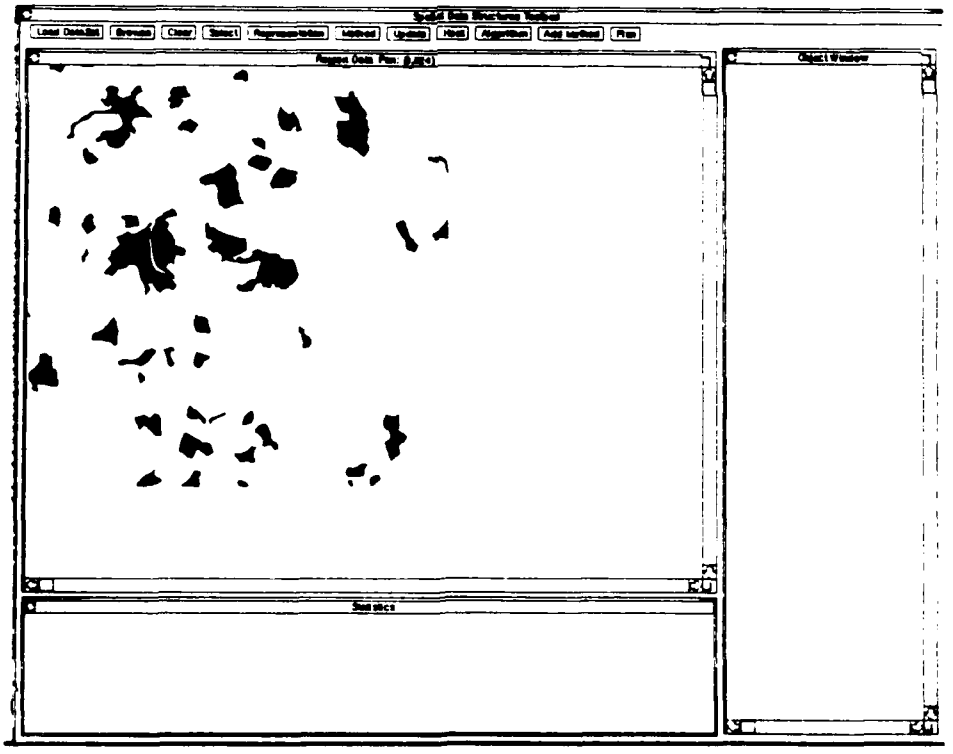


Figure 8-8: Union of Datasets 5A and 3I

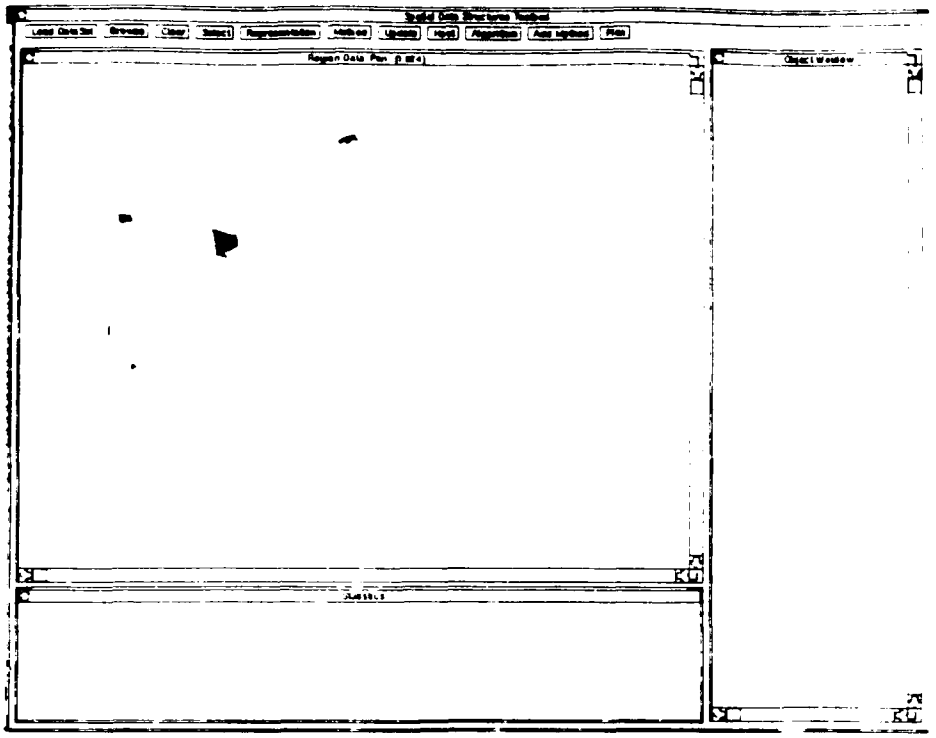


Figure 8-9: Intersection of Datasets 6A and 3I

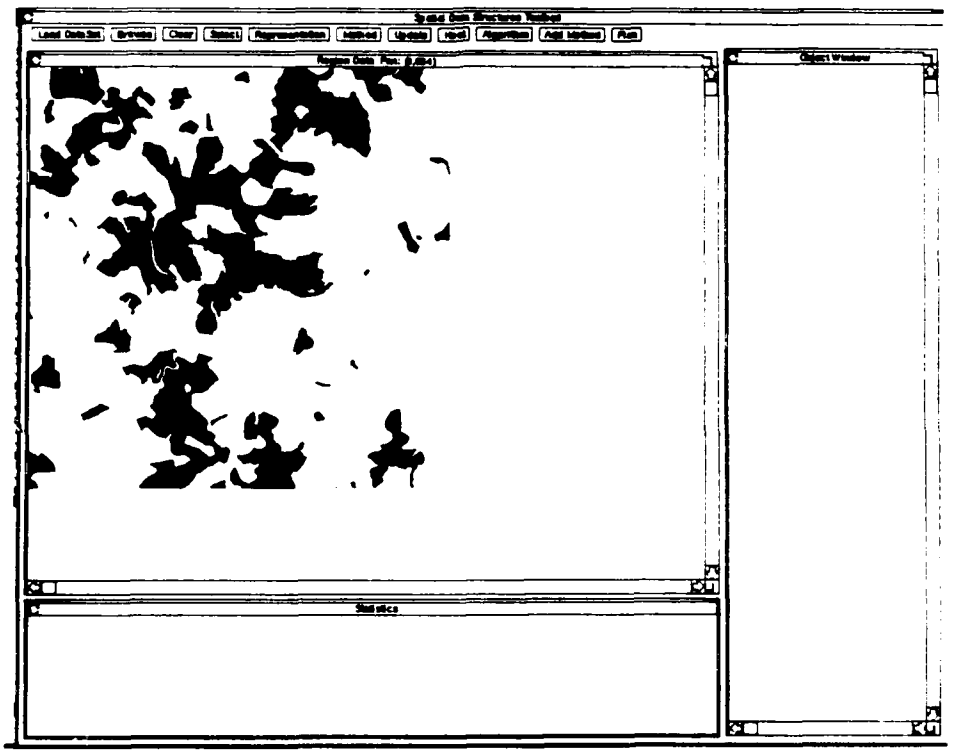


Figure 8-10: Union of Datasets 6A and 3I

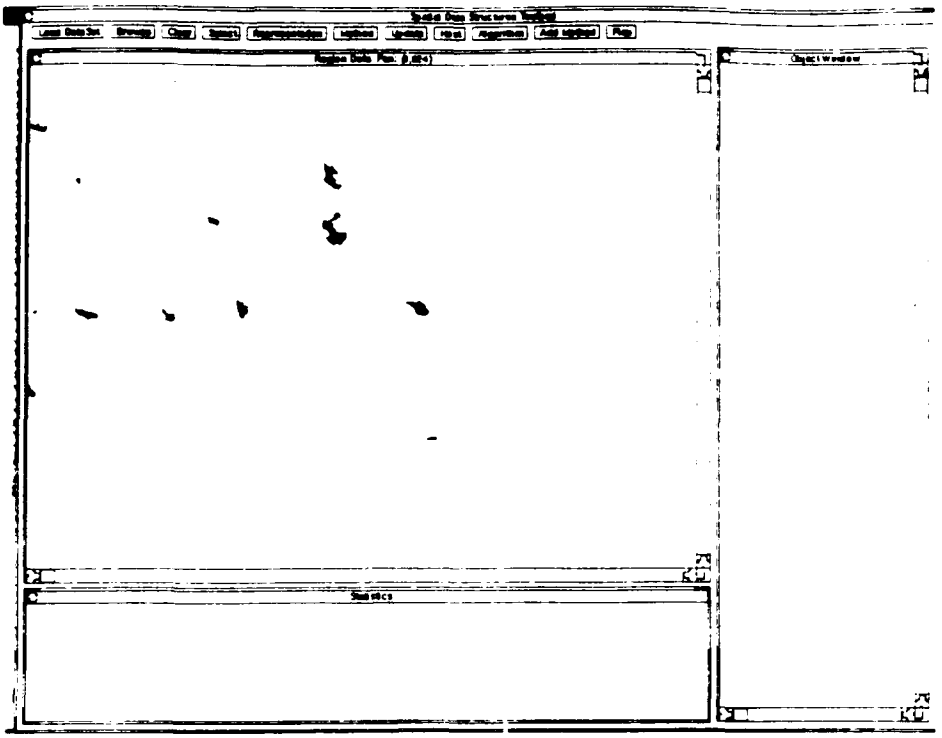


Figure 8-11: Intersection of Datasets 4A and 6I

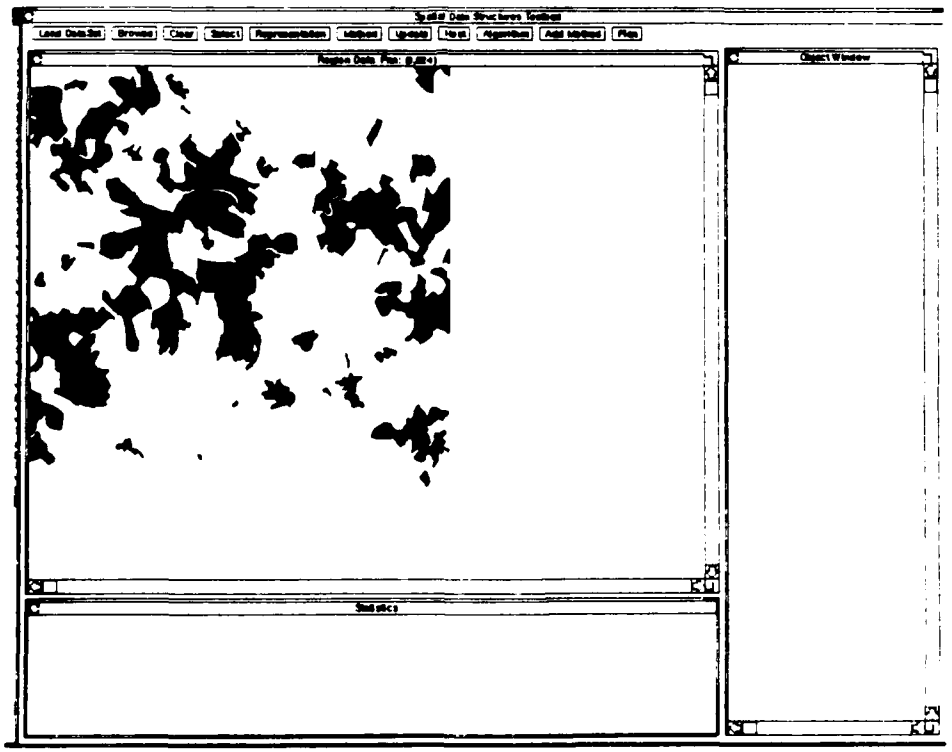


Figure 8-12: Union of Datasets 4A and 6I

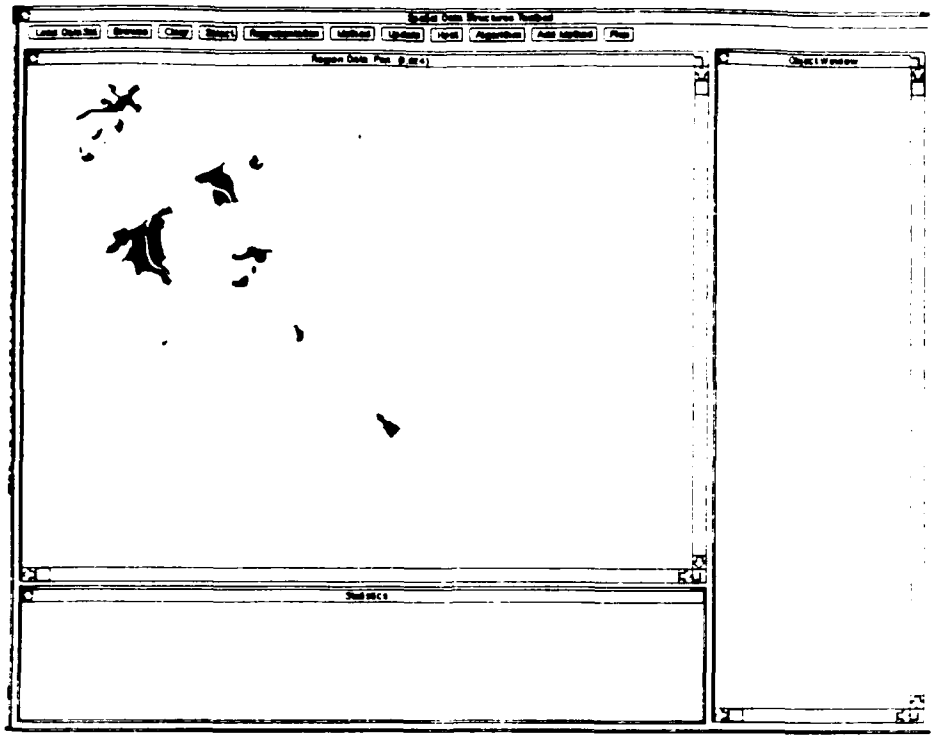


Figure 8-13: Intersection of Datasets 5A and 6I

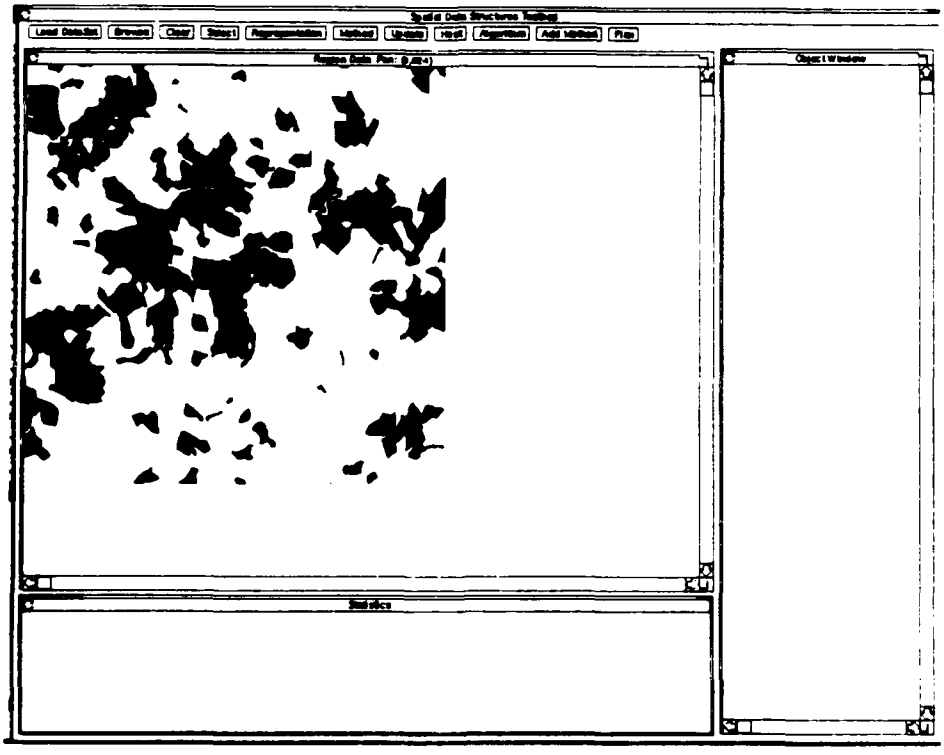


Figure 8-14: Union of Datasets 5A and 6I

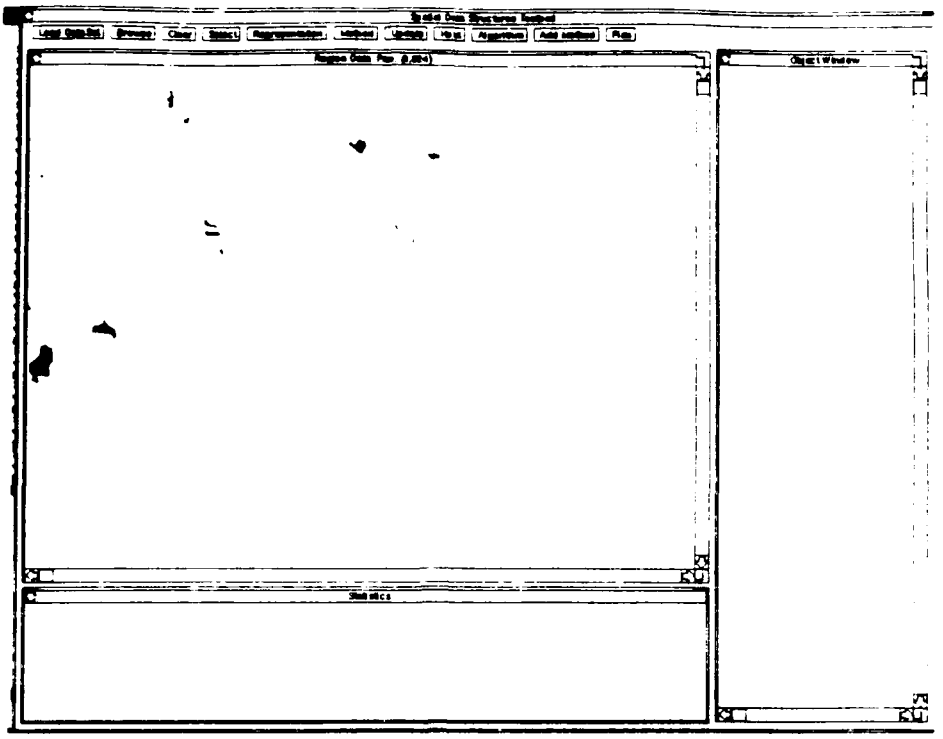


Figure 8-15: Intersection of Datasets 7A and 3I

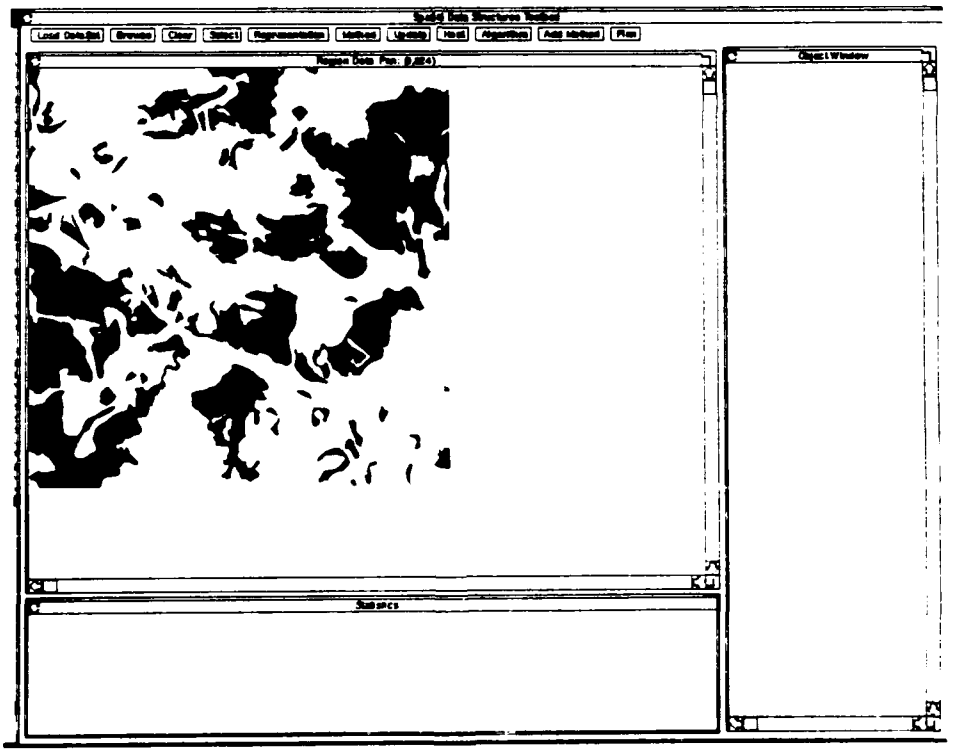


Figure 8-16: Union of Datasets 7A and 3I

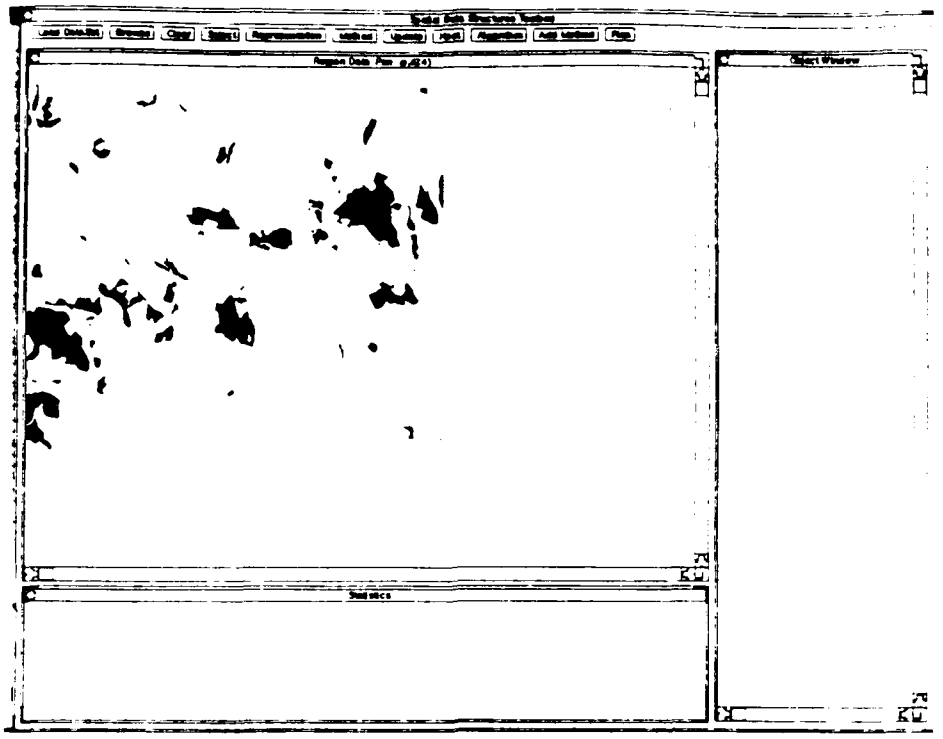


Figure 8-17: Intersection of Datasets 7A and 6I

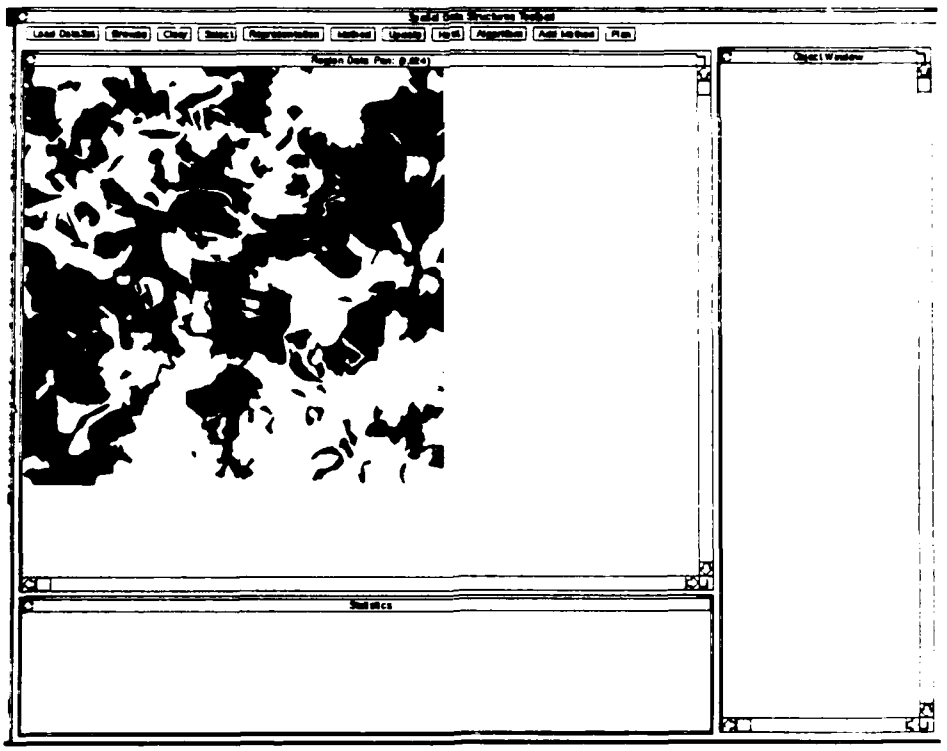


Figure 8-18: Union of Datasets 7A and 6I

Bibliography

[Bass 78] Basse, S, "Computer Algorithms: Introduction to Design and Analysis," Addison Wesley, Reading, Massachusetts, 1978.

[Barr 78] Barrow, H. G., et. al. (1978) "Parametric correspondence and chamfer matching: two new techniques for image matching," Proc., DARPA IU Workshop, pp. 21-27.

[Bobr 88] Bobrow, D., DeMichiel, L., Gabriel, R., Keene, S., Kiczales, G., Moon, D., (1988) "Common Lisp Object System Specification," Xerox Corp.

[Bres 65] Bresenham, J., "Algorithm for computer control of digital plotter," *IBM Syst. J.*, 4 (1) 1965, pp. 25-30.

[Cecc 87] Ceccatto, H. and B. Huberman, "The Complexity of Hierarchical Systems," to appear in *Phys. Rev. A*, (Dec. 1987).

[Fole 82] Foley, J., and Van Dam, A., "Fundamentals of Interactive Computer Graphics," Addison-Wesley Pub. Co., Mass, 1982.

[Gras 86] Grassberger, P., "How to Measure Self-Generated Complexity," *Physica*, 140A, pp. 319-325 (1986).

[Gren 69] Grenander, U., "Foundations of Pattern Analysis," *Quarterly of Applied Math*, 27, pp. 2-55, (1969).

[Gren 70] Grenander, U., "A Unified Approach to Pattern Analysis," *Advances in Computers*, 10, pp. 175-216, (1970).

[Hube 86] Huberman, B. and T. Hogg, "Complexity and Adaption," *Physica*, 22D, pp. 376-384 (1986).

[Levi 85] Levine, M. D., "Vision in Man and Machine," McGraw-Hill, New York, 1985.

[Prep 85] Preparata, F. P., and Shamos, M. I., (1985) "Computational Geometry, An Introduction," Springer-Verlag, New York.

[Shan 49] Shannon, C., and W. Weaver, "The Mathematical Theory of Communication," *Bell Syst. Tech. J.*, 27, pp. 379-423 (1949).

[Weil 80] Weiler, K., (1980) "Polygon comparison using a graph representation," Computer Graphics, 14, pp. 10-18.